The Journal of Systems & Software 198 (2023) 111605

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A component framework for the runtime enforcement of safety properties[☆]

Silvia Bonfanti [a], Elvinia Riccobene [b], Patrizia Scandurra [a,*]

[a] DIGIP, University of Bergamo, Viale Marconi, 5, Dalmine, 24044, Italy
[b] Università degli Studi di Milano, via Celoria, 18, Milan, 20135, Italy

A R T I C L E   I N F O

Article history:
Received 30 March 2022
Received in revised form 22 December 2022
Accepted 28 December 2022
Available online 4 January 2023

Keywords:
Runtime safety enforcement
Self-adaptation
MAPE-K
Runtime models
Abstract State Machines@run.time

A B S T R A C T

Safety assurance of a complex system cannot be completely ensured at design/development time since most uncertainties and unknowns are revealed when the system is deployed in a real environment. Safety assurance at runtime can be addressed by using models formalizing those safety assertions the system has to guarantee during operation, and specifying enforcement strategies aimed at preserving or eventually restoring safety.

This paper presents an approach to runtime safety enforcement of software systems based on the *MAPE-K control loop architecture* for system monitoring and control, and on the *State Machine* as runtime model to specify safety assertions and enforcement strategies for steering the correct system behavior. The enforcer software is designed to act as a proxy system which wraps around the software system to realize safety enforcement, both as *black-box enforcement* on unsafe I/O events and as *gray-box enforcement* on unsafe internal system changes. The proposed approach is supported by a component framework called RSE (*Runtime Safety Enforcement*) that is here illustrated by means of two real case studies in the health-care domain.

1. Introduction

Technological progress opens up opportunities for the creation of new business and digital ecosystems, but it also implies more demanding safety and security requirements of software systems to assure.

Nowadays software is used to control most systems, sometime called *software-enabled systems*, including physical systems under human controls, that could involve potentially large and even catastrophic losses (Leveson, 2020). Software system can be shown to be safe during the design/development stage by testing, simulation, or formal verification making explicit assumptions about the environment in which the system will execute. However, the safe behavior of a software system under certain circumstances cannot be completely ascertained at design/development time without deploying it in a real environment. It is fairly well known that it is necessary to deal with the assurance process of software systems also during the operational stage, when all relevant uncertainties and unknowns caused by the interactions of the system with their users and the environment can be detected and resolved (Lutz, 2000; Trapp and Schneider, 2014; Calinescu et al., 2018). So, controlling potentially unsafe systems requires approaches that combine development-time evidence with runtime evidence that the system continues to safely achieve its goals (Calinescu et al., 2018; Bennaceur et al., 2019).

Runtime verification and monitoring techniques (Calinescu and Kikuchi, 2011) have been developed in the past to address the problem of guaranteeing system safety at runtime. However, these approaches generally focus on the *oracle problem*, namely assigning verdicts to a system execution for compliance against policies formulated by means of an abstract model (e.g., in terms of temporal logic formulas). Differently, *runtime enforcement* (Falcone et al., 2018) focuses on preventing possible unsafe sequences of events and steering correct system executions, by possibly modifying or avoiding the system execution. Some *runtime enforcement* techniques have been used (see Falcone et al., 2018; Andersson et al., 2017; de Niz et al., 2018, to name a few) to modify the runtime behavior of a software system, forcing the system to satisfy a set of safety assertions, thus steering the behavior of the system to stay within its safe regions (Falcone et al., 2011) (a software system can work safely only in certain regions of its state space (He and Schumann, 2020; Andersson et al., 2020)).

The enforcement mechanism is usually synthesized according to a given automata-based formal specification, which treats the target system as a *black-box* by observing mainly input/output events. A *white-box* approach, that uses a complete knowledge

[☆] Editor: Prof. Raffaela Mirandola.
[*] Corresponding author.
E-mail address: patrizia.scandurra@unibg.it (P. Scandurra).

https://doi.org/10.1016/j.jss.2022.111605
0164-1212/© 2022 Elsevier Inc. All rights reserved.

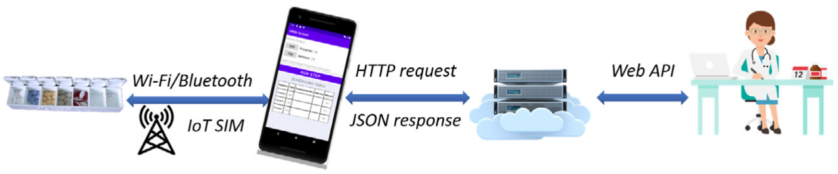

Fig. 1. MRM system: high-level architecture.

of the system and its external interactions, would suffer from problems of scalability and performance. A *gray-box* approach, instead, would be more effective; it could observe specific system's operational changes by probing and effecting interfaces provided by the target system, and it could then compute safety-related compensatory actions. However, the adoption of gray-box enforcement mechanisms has not been much explored yet (Falcone et al., 2018).

We here propose an approach and a supporting framework, called *RSE* (*Runtime Safety Enforcement)*, that allow both I/O sanitization in the black-box manner and safety enforcement in a gray-box way. In developing this approach, we leverage the use of software architecture-based self-adaptation (Garlan et al., 2009) and models@run.time (Calinescu and Kikuchi, 2011). By monitoring I/O events or probing them by suitable interfaces of the managed system, the proposed enforcement mechanism can detect operational changes that might lead to potential safety violations and proactively actuate an enforcement strategy as dictated by a runtime enforcement model. The *Abstract State Machine* (ASM) (Börger and Raschke, 2018) formal method is used to specify enforcement models. ASMs are used at runtime (by mean of the ASM@run.time (Riccobene and Scandurra, 2020b) engine) to detect safety assertions violation and whether/which reconfiguration actions are required to guarantee compliance with the safety goals. In the gray-box use, the enforcement mechanism is engineered as an autonomic manager that through a feedback control loop *MAPE-K (Monitor-Analyze-Plan-Execute over a Knowledge base)* (Kephart and Chess, 2003) wraps around the target system. It uses an ASM runtime enforcement model to plan an enforcement strategy and therefore steers the correct system behavior by *adapting* unsafe system changes on the base of the planned strategy.

To illustrate RSE (approach and framework), we consider two case-studies in the health-care domain, namely a *Medicine Reminder and Monitoring System* (MRM) for securing patient safety through a software enforcer that remotely controls a smart medicine dispenser (Bombarda et al., 2019), and the *Mechanical Ventilator Milano* (MVM) (Abba et al., 2021) for securing patients on mechanical ventilation in case of apnea lag and lung stress. The first case-study is a realistic example of a medicine dispenser device, the second is a real system developed during COVID-19 pandemic.

The main contributions of this paper in the context of system runtime enforcement are the following:

- the formal description of the two approaches, black-box and gray-box enforcement, of RSE;
- the process to apply the RSE approach on a target system;
- the instantiation of the steps of the RSE process on two case studies from the medical domain;
- the evaluation of the RSE in terms of soundness and computation overhead.

This paper widely extends previously results presented in Bonfanti et al. (2021), and Riccobene and Scandurra (2020a). The idea of exploiting ASMs and ASM@runtime for runtime enforcement is shortly presented in Riccobene and Scandurra (2020a), in a very informal and preliminary way. In Bonfanti et al. (2021) we present only the gray-box approach but without any formalization and showing only a partial application on one of the two case studies reported here. The formal description of the black- and gray-box approaches in terms of ASM enforcement models, the RSE process, an additional case study, the complete application of the RSE to the two case studies and the framework evaluation, are novel contributions.

The paper is organized as follows. Section 2 presents the two case studies used throughout the paper as illustrative examples. Section 3 provides some preliminaries on runtime enforcement mechanisms and basic concepts on the ASM formal method. Section 4 presents the enforcement mechanisms supported by the RSE approach and the use of ASMs as runtime models of the enforcement schemas, including the conceptual and theoretical foundations. Section 5 describes the operational steps of the RSE process. Section 6 illustrates the RSE step of defining the enforcement strategies on the two running case studies, while formal specification of enforcement strategies in terms of ASM models is described in Section 7. Section 8 provides implementation details of the proposed RSE component framework, while Section 9.1 shows the execution of RSE on the two case studies, and reports the results of validation experiments. Section 10 highlights related work, and Section 9.2 discusses some threats to the RSE validity. Finally, Section 11 concludes the paper.

2. Illustrative case studies

In this section, we introduce the two case studies, MRM (Medicine Reminder and Monitoring System) and MVM (Mechanical Ventilator Milano), to which we have applied the RSE process. The first case-study is a realistic example of a smart medicine dispenser, the second is a real system developed during COVID-19 pandemic.

2.1. MRM

The MRM system remotely controls a smart medicine dispenser and secures patient safety. The proposed example is inspired by real smart drug systems available on the market and consists of (see Fig. 1) an electronic pill reminder&dispenser (a pill box), a remote control app (e.g., a mobile app running on an Android smartphone) that monitors and controls the pill box (via a local wireless connection, e.g. WiFi or Bluetooth, when the caregiver is in proximity, or via IoT SIM card when the caregiver is away from the patient's home), and an health-care information system on Cloud implementing value-added and persistence services related to a person's health record, treatment and medicine prescriptions as inserted by the doctor via a web app. The remote control app is responsible for downloading/uploading patient's information from/to the user's electronic health record on Cloud. The app is also responsible for the correct pill box initialization once filled and plugged into a home wall outlet, and for dynamically enforcing the pill box re-configuration to ensure the patient safety about medicines intake. If errors happen during pill box initialization, it is put into *fail-safe* configuration: the system is locked and an alarm is displayed. The user has to take corrective actions, e.g., try again the connection of the application or contact the service center.

Initially, the caregiver or the patient through the mobile app downloads a drug file record (e.g., a JSON file) containing all information about the medicines prescribed by the doctor: medicine name, number of doses per day, time schedule, minimum separation (in terms of time) from the medicine M to the interferer *N* and between the same medicine, and delta time added to the original time schedule to remember the medicine again if a dose is missed. Then, the user has to manually fill the pill box's compartment with the medicines (one medicine type per each compartment) on a daily basis according to the given prescription for the overall treatment duration. Once the medicines have been added into the pill box and the pill box actual configuration has been checked against the prescription via the remote control app, the patient is notified by the pill box when a medicine has to be taken. At the programmed time of a medicine, a notification is sent to the mobile remote control app, an audible alarm of the pill box sounds, a red led, corresponding to the compartment where the medicine to be taken is located, is turned on and the compartment is unlocked. The patient/caregiver has to open and then close the compartment to report to the remote control app that the medicine was effectively taken. If after 10 min from the expected time the medicine is not taken, the red led starts flashing for 10 min further, after which the pill is considered missed by the system.

In case of a missed pill, drug reminder usually notifies the caregiver only. In this example, instead, we assume the remote control app is engineered smarter with a software enforcer that assists patients and caregivers in the medicine intake by reconfiguring the pill box automatically. Intuitively, a missed medicine can be re-scheduled later in time, and in case of delayed medicine intake, it must be guaranteed that the next medicines are taken without drug interference.

2.2. MVM

MVM[1] (Abba et al., 2021) is a device developed to provide ventilation support for patients in intensive therapy, which require mechanical ventilation. It is based on pressure ventilation therapy, and it supports two ventilation modes: *Pressure Controlled Ventilation* (PCV) and *Pressure Support Ventilation* (PSV). PCV mode is used for patients that are not able to start breathing on their own. The duration of the respiratory cycle is kept constant and set by the doctor; and the pressure changes between the target inspiratory pressure and the positive end-expiratory pressure. When the MVM detects a sudden pressure drop within the trigger window during expiration, a new inspiration is initiated even if the set time has not elapsed. In PSV mode, the respiratory cycle is controlled by the patient, and MVM partially takes over the work of breathing. A new inspiration is initiated when a sudden pressure drop occurs, while expiration starts when the patient's inspiratory flow drops below a set fraction of the peak flow. If a new inspiratory phase is not detected within a certain amount of time (apnea lag), MVM automatically switches to the PCV mode because it is assumed that the patient is not able to breathe alone.

MVM has two valves to enter/exit the air, one input valve and one output valve. The input valve opens in inspiration phase and closes in expiration phase, while the output valve opens and closes oppositely.

Before starting the ventilation, the MVM controller passes through three phases. The *start-up* in which the controller is initialized with default parameters, *self-test* which ensures that the hardware is fully functional, and *ventilation off* in which the controller is ready for ventilation when requested. When in ventilation off, before ventilating the patient, if user inputs are not correct, e.g., within their upper and lower limits, the system is put into *fail-safe* configuration: the ventilation does not start and an alarm is raised, till user sets them correctly.

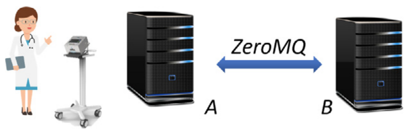

Fig. 2. MVM system: high-level architecture.

During ventilation, a *safe-mode* configuration (a further *fail-safe* configuration) is reached when errors not automatically manageable occur. In this configuration the input valve is closed and the output valve is opened to allow the patient to breathe thanks to two relief valves.

Since it is not possible to test the proposed enforcement approach on the real system, we have decided to use the configuration shown in Fig. 2. We run the MVM code on machine A and the remote control system on machine B. The communication is implemented in ZeroMQ.[2] The remote control system monitors the behavior of the MVM and, in case of apnea not detected by the MVM controller (because e.g. it is temporarily in error) or lung stress (the MVM is ventilating in PCV, but PSV is the better ventilation mode because the patient continuously triggers breaths), changes the ventilation mode in order to provide better comfort to the patient.

3. Preliminaries and definitions

3.1. Self-adaptation and runtime models

A MAPE-K feedback control loop (Kephart and Chess, 2003) is an architectural pattern to structure the adaptation layer of a self-adaptive software system. It is conceived as a sequential execution of four MAPE components: Monitor, Analyze, Plan, and Execute over a shared Knowledge. A self-adaptive system usually relies on the use of runtime models as part of the knowledge base. A runtime model acts as a first-class runtime abstraction of the managed system, the environment, the requirements (the adaptation goals) or of any data insights that can be used by the feedback loop for making adaptation decisions.

A causal connection (Bennaceur et al., 2014; Tendeloo et al., 2019) must be realized to link and synchronize the state of a runtime model with that of the managed system and its environment. Usually, an acceptable discrepancy between the two that must not invalidate the adaptation process of the managing system is allowed. The discrepancy may be temporal causing a delay in the updates of the model, quantitative in the sensed and processed data, or be of any other form.

3.2. Runtime enforcement concepts and definitions

We here provide some preliminary concepts and definitions about safety enforcement. Firstly, consider Fig. 3 for the depiction of the following:

Definition 1 (*System Operational Change*).
A system S operates in a physical environment Env. A context $C \subset Env$ of a system S represents the environmental entities that interact with S and influence its behavior. During operation, the

[1] https://www.vexos.com/mvm-ventilator

[2] https://zeromq.org/

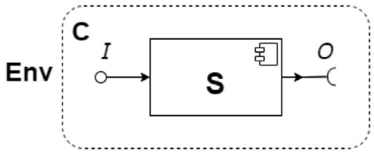

Fig. 3. A system and its context. The circle (or ball) indicates input events that the system can handle; the semi-circle (or socket) indicates output events from the system.

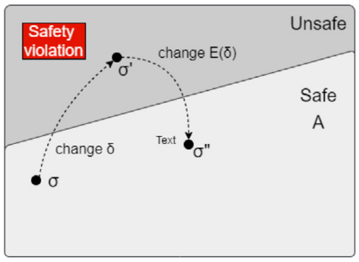

Fig. 4. Safe/Unsafe/Safety violation state space.

system perceives parts of C through its input interface (or set of input events) I and reacts accordingly by affecting parts of C through its output interface (or set of output events) O.

Note that the context C may also include human actors or adjacent systems.

In the following, we denote by $\delta(I, S, O)$ (δ in brief) an *operational change* made by S processing input I and providing output O.[3]

As in Andersson et al. (2020), we adopt the following simplified notion of system state.

Definition 2 (*System State*)**.** A *system state* is a vector σ belonging to some n-dimensional state space Σ representing the collection of variables required for describing the attributes of a system and its context.

We classify the system state space into safe, unsafe, and safety violation sub-spaces.

Definition 3 (*Safe System Region*)**.** A system *safe region* is a subset of the system state space Σ consisting of all those states where the system is safe since a given set of safety assertions A is guaranteed. We denote by $A(\Sigma)$ this subset of safe states.

Definition 4 (*Unsafe System Region*)**.** A system *unsafe region* is a subset of the system state space Σ consisting of all those states where the system might be unsafe since a given set of safety assertions A could be not guaranteed. We denote by $\Sigma \setminus A(\Sigma)$ this subset of unsafe states.

Definition 5 (*Safety Violation System Region*)**.** A system *Safety Violation region* is a subset of the system state space Σ consisting of all those states where the safety assertions A are not guaranteed. We denote by $\neg A(\Sigma)$ this subset of safety violation states.

The set $\neg A(\Sigma)$ is a subset of $\Sigma \setminus A(\Sigma)$. Fig. 4 shows the three possible regions in which a system can be: *safe* if a system is in a safe state; *unsafe* if it is in a state with possibility of violations; *safety violation* if a system is in a state where safety assertions are violated.

[3] We use short notation $\delta(I)$ or $\delta(O)$ when only interested on the I/O of δ.

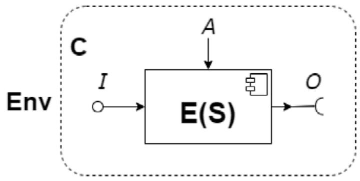

Fig. 5. Enforcer software.

We now introduce the concept of *safe and unsafe* step of the system depending whether it remains or not inside the boundary of the safe regions of its state space after an execution step, upon the occurrence of events or operational changes.

Definition 6 (*System Safe Step*)**.** Given OC the set of operational changes that may affect a system or its environment, we say that a system change step $(\sigma, \delta, \sigma')$ is safe with respect to an operational change $\delta \in OC$ iff $\sigma' \in A(\Sigma)$, being $\sigma \in A(\Sigma)$.

Definition 7 (*System Unsafe Step*)**.** Given OC the set of operational changes that may affect a system or its environment, we say that a system change step $(\sigma, \delta, \sigma')$ is unsafe with respect to an operational change $\delta \in OC$ iff $\sigma' \in \Sigma \setminus A(\Sigma)$, being $\sigma \in A(\Sigma)$.

To repair undesired operational changes that bring the system to the unsafe region, a (software) enforcer E (see Fig. 5) can be used to check if it is possible to steer S to stay in the safe region. The enforcer E should ensure that the system will never remain in an unsafe state (i.e., the system could temporarily enter an unsafe state, but the enforcer forces it to return to a safe state).

Definition 8 (*System Enforcer*)**.** An enforcer E of a target system S is a software component that steers the runtime behavior of S to ensure satisfaction of the safety assertions A in the context C. Formally: if $(\sigma, \delta, \sigma')$ is an unsafe step of S ($\sigma' \notin A(\Sigma)$), E operates an adaptation change $E(\delta)$ of the operational change of S such that S makes a step $(\sigma', E(\delta), \sigma'')$, where $\sigma'' \in A(\Sigma)$.

Core to the enforcement process is a runtime *enforcement model* m_S used for reasoning whether adjustments actions are required or not to enforce safety properties. Essentially, it is an automaton-based mechanism endowed with an internal memory (Falcone et al., 2011) used by the actual software enforcer to specify the enforcement logic. This enforcement model is continuously updated with runtime data extracted by monitoring the running system to reflect the system status changes.

We assume there exists a *causal connection* (Bennaceur et al., 2014; Tendeloo et al., 2019) between the enforcement model m_S and the running system S (or an observed part of it): m_S (1) is causally connected with events (input I and output O) of S, and (2) can be causally connected with change events δ of S. In the first case, E uses m_S as *oracle* model to make I/O sanitization (i.e., black-box enforcement); m_S only provides a safe/unsafe verdict by checking safety violations. In the second case, m_S has an embedded enforcement logic that E uses to plan adjustment actions of functionality of S (for gray-box enforcement). A more precise description of the enforcement mechanisms is given in Section 4 in the context of the RSE framework.

3.3. Abstract state machines

ASMs (Börger and Raschke, 2018) are an extension of Finite State Machines where unstructured control states are replaced by *states* comprising arbitrary complex data (i.e., domains of objects with functions defined on them), and *transitions* are expressed by transition rules describing how the data (state function values saved into *locations*) change from one state to the next.

ASM models can be read as "pseudocode over abstract data" which comes with a *well-defined semantics*: at each computation step, all transition rules are executed in parallel by leading to simultaneous (consistent) updates of a number of locations. There is a limited but powerful set of *rule constructors* to express: guarded actions (`if-then`, `switch-case`), simultaneous parallel actions (`par`), sequential actions (`seq`), nondeterminism (existential quantification `choose`), and unrestricted synchronous parallelism (universal quantification `forall`).

An ASM *run* is defined as a finite or infinite sequence of states, starting from an initial state and each state obtained from the previous one by firing the unique *main rule* which in turn could fire other transitions rules.

Model interface with its environment is specified in terms of *monitored* (written by the environment and read by the machine) and *out* (written by the machine and read by the environment) functions. It is also possible to specify state *invariants* that are first order formulas that must be true in each computational state. A set of safety assertions can be specified as model invariants, and a model state is *safe* if state invariants are satisfied.

Without going into the details that will be presented afterwards, Code 1 reports an excerpt of the ASM *safePillbox*, which operates as a gray-box enforcement model of the MRM system. The section `signature` declares all functions of the model, among which those specifying the model interface with its environment (i.e., `monitored` and `out`). The `main rule` is, at each state, the starting point of the computation; it, in turns, invokes all the other transitions rules (e.g., `r_enforce`). The section `default init` defines the initial values for the *controlled* functions (updated only by the machine).

```
asm safePillbox
import StandardLibrary
import pillbox_sanitiser
signature:
    monitored isPillMissed: Compartment −> Boolean
    monitored pillTakenWithDelay: Compartment −> Boolean
    monitored actual_time_consumption: Compartment −> Seq(Natural)
    out setNewTime: Compartment −> Boolean
    out setOriginalTime: Compartment −> Boolean
    out newTime: Compartment −> Natural
    out skipNextPill: Prod(Compartment,Compartment) −> Boolean
    monitored systemTime: Natural
definitions:
    rule r_enforce = forall \$compartment in Compartment do
        par r_pillOnTime[\$compartment] r_noOverlapping[\$compartment]
               endpar ...
    main rule r_Main = if state = INIT then r_INIT[] else r_enforce[] endif
default init s0:
    function state = INIT
    function medicine_list = ["fosamax","moment"]
    function amount(\$medicine in String) = switch(\$medicine)
                                                case "moment" : 2n
                                                case "fosamax" : 1n
                                          endswitch ...
```

Code 1: SafePillbox ASM model

Recently, a runtime simulation platform has been developed for ASMs (Riccobene and Scandurra, 2020b) within the ASMETA (ASM mETAmodeling) analysis toolset (Anon, 2022) – a set of modeling and V&V tools for the ASM formal method – to check safety assertions of software systems at runtime and support on-the-fly changes of safety assertions. The platform exploits the concept of executable ASM models and it is based on the `AsmetaS@run.time` simulator to handle (including model-roll back capabilities) an ASM model as a living/runtime model possibly executing in tandem with a prototype/real system and provide formal support for system properties assurance (Arcaini et al., 2021).

ASMs have been already used to provide formal specification of MAPE loops for adaptation concerns (Arcaini et al., 2017a, 2015) and for solving interfering adaptation goals (Arcaini et al., 2020). By exploiting the notion of *multi-agent ASM* – where each agent of the predefined set of *Agents* executes its own ASM in parallel with other agents' ASMs –, the definition of *self-adaptive ASMs* has been given (Arcaini et al., 2017a, 2015) to provide formal specification of MAPE loops for adaptation concerns: the set *Agents* is the disjoint union of a set of *managing agents* encapsulating the logic of self-adaptation, and a set of *managed agents* encapsulating the system's application logic.

Here, ASMs are used as runtime models to specify the enforcement strategy of the enforcer software. The main characteristics of ASMs suitable for the enforcement mechanisms are: (1) due to their *pseudo-code format*, ASMs can be easily understood by practitioners and can be used for high-level programming; (2) ASMs offer a precise system specification at any desired *level of abstraction*; (3) ASMs are *executable models*, so they can be co-executed with system low-level implementations (Riccobene and Scandurra, 2014); (4) the concept of ASM *module*, i.e., an ASM without the main firing rule, facilitates model scalability and separation of concerns, so facing the complexity of big systems specification; (5) ASMs support multi-agent compositions, which allows for *modeling distributed and decentralized software systems* (Arcaini et al., 2017a).

4. The RSE approach to safety enforcement

This section presents the enforcement mechanisms supported by the RSE framework and the use of ASMs as runtime enforcement models.

4.1. Reference architectural schemes for the RSE enforcement mechanisms

RSE supports enforcement mechanisms to realize input/output sanitization by suppressing/adjusting the system input/output before their use (*black-box enforcement*), and to steer the system execution to stay within safe regions by monitoring and (eventually) adapting the system behavior through effectors/actuators (*gray-box enforcement*). Fig. 6 shows the architectural schemes of these three types of enforcement.

4.1.1. Black-box enforcement

This enforcement approach reflects, in our framework, the input/output sanitization as defined in Falcone et al. (2018).

Input sanitization. In order to protect the system from its (untrusted) environment, all inputs for the system are checked first by the enforcer that filters out those that could harm the system. The enforcer uses a runtime model m_S representing the state (or part of it) of the target system and its operational environment, including safety assertions A on the input events I that must hold for I to be safe; m_S works as *oracle model* for E to provide a safe/unsafe verdict, or to make input sanitization.

Formally: if $(\sigma, \delta(I), \sigma')$ is a state change of S and $\sigma' \in \Sigma \setminus A(\Sigma)$, then (*i*) E forces S to remain in the same state without executing δ and $\sigma' = \sigma$ or (*ii*) E sanitizes I in $I' = E(I)$ such that $(\sigma, \delta(I'), \sigma'')$ is a safe step of S (i.e., $\sigma'' \in A(\Sigma)$).

Output sanitization. The enforcer is used to check the system outputs to filter or transform them, thus protecting the environment from the system itself. A runtime model m_S is used by E as oracle model representing the state (or part of it) of the target system and its operational environment, including safety assertions A on the output events O that must hold for O to be safe.

Formally: if $(\sigma, \delta, \sigma')$ is a state change of S and $\sigma' \in \Sigma \setminus A(\Sigma)$, then (*i*) E forces S to remain in the same state without executing δ and $\sigma' = \sigma$; or (*ii*) E sanitizes O in $O' = E(O)$ such that $(\sigma, \delta(O'), \sigma'')$ is a safe step of S.

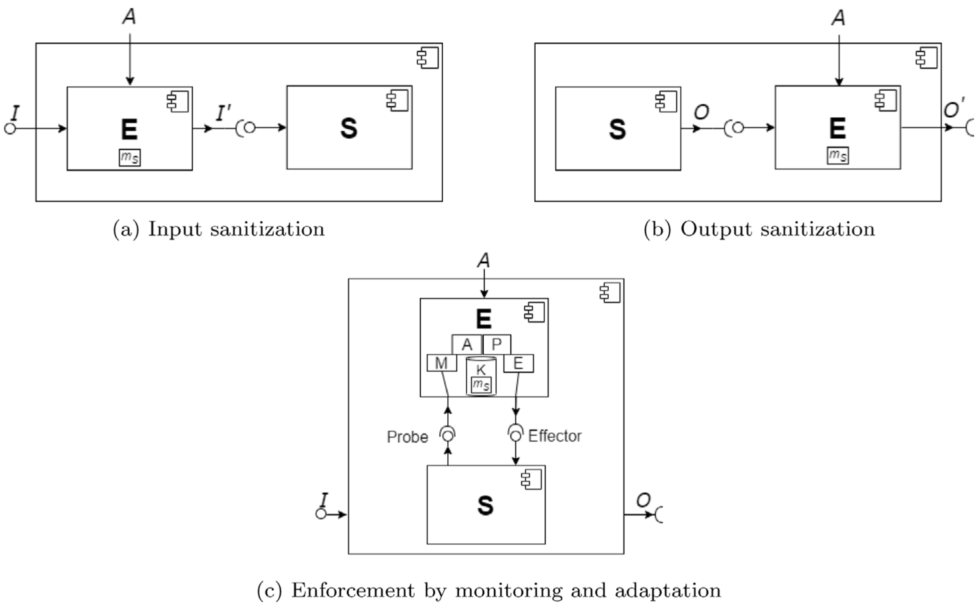

Fig. 6. Safety enforcement mechanisms.

4.1.2. Gray-box enforcement

This enforcement mechanism is obtained by monitoring and adaptation. The enforcer E manages the target system S through a MAPE-K feedback loop in an environment Env. Safety-critical actions or relevant state changes of S are monitored by the enforcer (through sensors/probes *Probe* of the system, see Fig. 6(c)) that can intervene by adapting or modifying the system changes (through effectors/actuators *Effector* of the system). The runtime *enforcement model* m_S is part of the knowledge K and is used for evaluating whether/which reconfiguration actions are required to guarantee safety properties A. The enforcement model m_S is continuously fed up with *Probe* data extracted from the monitored system S about state changes of the system and its context C, and executed at each feedback loop step to analyze whether the safety assertions A continue to be satisfied; whenever this is no longer the case, appropriate system changes are planned and executed by the enforcer. So the enforcer plays the role of autonomic manager of the enforcement process.

Formally: if $(\sigma, \delta, \sigma')$ is a state change of S and $\sigma' \in \Sigma \setminus A(\Sigma)$, then E adapts the behavior of S in δ' such that $(\sigma', \delta', \sigma'')$ is a safe step of S.

4.2. ASMs as runtime enforcement models within RSE

By exploiting the concept of ASM@run.time, an ASM model can be defined and adopted as runtime model m_S of the proposed enforcement mechanisms. ASM@run.time also incorporates mechanisms to rollback an ASM runtime model to its previous safe state before processing the model input causing a failure (like invariant violations).

ASMs runtime models can operate within the RSE framework as follows. The enforcer monitors the target system S by watching its relevant input/output or probe events. When a new event occurs that may change the state of the system, the enforcer observes the ASM m_S execution first and then, if safety assertion violation may occur: *(i)* it prevents the system change — m_S is used as oracle model in the black-box enforcement with system blocking effect; or *(ii)* it adjusts the system change — m_S is used as gray-box enforcement model with system adjusting effect. This is better explained in the following.

ASM as black-box enforcement model. In this case, safety violations are not handled at the ASM model level, and the ASM works only as oracle emitting a verdict (safe or unsafe). Safety assertions are formally specified in terms of ASM *invariants*, that are, boolean predicates over ASM functions that must be true in any ASM state. Safety violation results in the ASM reaching a state where such invariants do not hold; in this case the model rolls back to its previous safe state. Safety assertions can evolve dynamically to incorporate, for example, at ASM model level new safety invariants coping with uncontrollable/emergent events not foreseen at design time. This approach is typically used for input/output sanitization with the enforcer filtering out input/output events without compensation actions. Concrete examples of ASM oracle models are given in Section 7 for the two case studies.

ASM as gray-box enforcement model. In this case safety violations are captured and compensation is planned at the ASM model level. Safety violations are specified as predicates over ASM state functions, and occur, in a negated form, as *guards* of ASM guarded transition rules representing enforcement operations – here called ASM *enforcement rules* – according to a specific enforcement logic. Safety violation therefore happens when enforcement rules can fire; the output (updates of out locations) of the ASM is used by the enforcer as a prescription/plan for adapting the monitored system S.

An enforcement rule has the form of an ECA (Event Condition Action) rule:

```
if $\neg(\alpha(e_p))$ then
    if $\rho(e_p)$ then  enforcement plan  endif
endif
```

If $\alpha \in A$ is a safety assumption, the rule fires when an intercepted probing event e_p can violate α, and a *safety enforceability* condition ρ holds – i.e., $\rho(e_p)$ guarantees the possibility to steer the system to a safe region again –, then suitable enforcement actions (i.e., the *enforcement plan*) are executed.

Concrete examples of ASM enforcement models are provided in Section 7 for the MRM and the MVM systems.

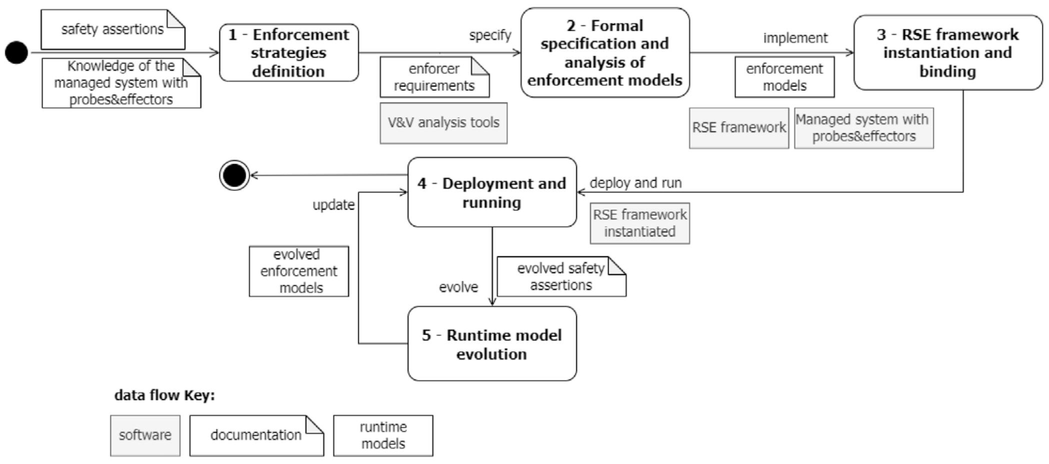
safety assertions
Knowledge of the managed system with probes&effectors
1 - Enforcement strategies definition
specify
enforcer requirements
V&V analysis tools
2 - Formal specification and analysis of enforcement models
implement
enforcement models
RSE framework
Managed system with probes&effectors
3 - RSE framework instantiation and binding
deploy and run
RSE framework instantiated
4 - Deployment and running
update
evolved enforcement models
evolve
evolved safety assertions
5 - Runtime model evolution
data flow Key:
software
documentation
runtime models

Fig. 7. The five stages of the RSE process.

5. RSE process

We here describe how to realize in practice the RSE enforcement approach for a given target system. The RSE process (see the UML-like activity diagram shown in Fig. 7) consists of five stages spanning the design, development, and operation phases of software life-cycle:

Stage 1 — Enforcement strategies definition. Formulation of the safety assertions, definition of I/O and probing/effecting interfaces of the target observed system, and definition of enforcement strategies for both black-box and gray-box enforcement — *@design.time*.

Stage 2 — Formal specification and analysis of enforcement models. Formal specification, validation and verification of enforcement models for the enforcement of the safety assertions expressed over the global state of the runtime models — *@design.time*.

Stage 3 — RSE framework instantiation and binding. Instantiation and binding of the enforcer component framework as wrapper causally connected with the target system and endowed with the runtime model(s) for safety enforcement — *@development.time*.

Stage 4 — Deployment and running. Deployment, set up, and running of the overall enforced system — *@run.time*.

Stage 5 — Runtime model evolution. Adaptation of safety assertions and enforcement rules to take into account new requirements — *@design.time or @run.time*.

All these stages, except the validation and verification results of stage 2 for the functional correctness of the enforcement models and stage 5, are illustrated in the next sections through the two running case studies. Regarding stage 2, we only present the enforcement models used @runtime. They have been developed, and formally validated and verified at design-time by means of the ASMETA toolset, to guarantee that they realize their functions correctly w.r.t. a set of requirements (*enforcer requirements*) the enforcement models should comply with. These analysis activities have been performed on the enforcement models for both the two case studies, however, related results are not reported here since out of the scope of this paper.

Regarding stage 5, RSE provides some support to realize requirements at runtime (RE@runtime (Kramer, 2020)) for the management of evolution changes in the enforcement goals. Essentially, the RSE framework supports on the fly changes of ASM model invariants for safety assertions in the black-box mechanism. Through a user-facing notation, users (system maintainers or supervisors) of the RSE installation can add/eliminate/modify ASM invariants of the runtime ASM model. This feature of the framework is part of the ASM@run.time engine and still in a prototype development phase (Riccobene and Scandurra, 2020b). The dynamic change of ASM enforcement rules in the gray-box mechanism is, instead, considered an open challenge in the runtime enforcement discipline and not yet supported by RSE, though feasible. New enforcement requirements may also imply changes in the I/O and probing/effecting interfaces used to observe the target system and also new sources of uncertainty at the level of the enforcement models. We postpone the management of these evolution changes as future work.

6. Enforcement strategies formulation

This section illustrates the stage 1 of the RSE process for the two running case studies.

6.1. MRM safety enforcement scenarios

Starting from the MRM case study description in Section 2.1, we have identified the enforcement scenarios for both black-box and gray-box enforcement, and we have specified safety assertions as shown in the following sections.

6.1.1. Black-box safety enforcement scenarios

The safety assertions we have identified guarantee the correct configuration of the MRM system based on the doctor prescription:

- PILL TYPE CONSISTENCY: pills added in the compartments of the pillbox must be consistent with those prescribed by the doctor;
- PILL AMOUNT CONSISTENCY: for each type of pill, the number of pills added in each compartment of the pillbox must be consistent with the number of pills prescribed by the doctor;
- PILL TIME CONSISTENCY: for each pill, the time schedule of the compartment's slots must be the same as that prescribed by the doctor;
- NO STATIC INTERFERENCE: for each pill, the time between its assumption and the assumption time of the next pill must be less than the minimum separation time of the two pills.

Table 1
Black-box safety enforcement scenarios MRM.

Scenario	Unsafe condition	Enforcement strategy	Enforcement plan
PB1	Pill type inconsistency	Lock administration	Fail-safe activation
PB2	Pills amount inconsistency	Lock administration	Fail-safe activation
PB3	Pills time inconsistency	Lock administration	Fail-safe activation
PB4	Static interferences	Lock administration	Fail-safe activation

Table 2
Gray-box safety enforcement scenarios MRM.

Scenario	Unsafe condition	Enforcement strategy	Enforcement plan
MP1	Pill missed	Postpone missed pill without overlapping	Re-schedule pill
MP2	Pill missed	Skip missed pill with overlapping	Skip current missed pill
LP	Pill taken later	Avoid pills overlapping	Skip next pill if it overlaps

Table 3
Black-box safety enforcement scenarios MVM.

Scenario	Unsafe condition	Enforcement strategy	Enforcement plan
APRange	Apnea lag out of range	Lock ventilation	Fail-safe activation
IE	I:E ratio	Lock ventilation	Fail-safe activation

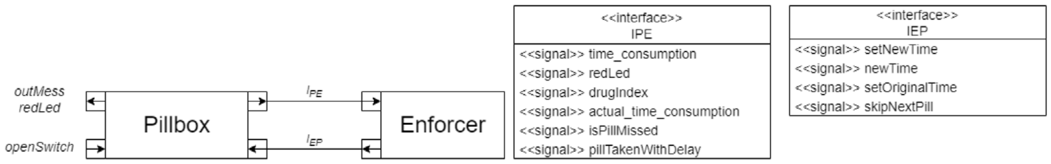

Fig. 8. MRM system and signals exchanged.

In case safety assertions are violated, the enforcer activates the *fail-safe* configuration: the system is locked and requires user intervention to restore the pills based on doctor prescription. We have listed the enforcement scenarios as reported in Table 1.

6.1.2. Gray-box safety enforcement scenarios

We have identified the following safety assertions that must be satisfied to guarantee that the MRM system operates in a safe way for the patient:

- PILL ON TIME: pills must be taken at the timing of medications within a delta time window;
- INTERFERING PILLS NEVER OVERLAP: consumption times of interfering pills must not overlap.

In case safety assertions are violated, we assume the enforcer automatically re-configures the medications according to the following *safety enforcement strategies*:

MISSED PILL (MP): A missed pill is re-scheduled by the enforcer by adding the delta time to the original time schedule only if the minimum separation time from the next medicines is observed, otherwise it is definitively missed.

LATE PILL (LP): Since from the scheduled time to the actual time consumption can take up to 20 min, the enforcer checks if the difference between the actual time consumption and the minimum separation from the next medicines is observed. If not, the next pill is skipped.

Table 2 reports the three safety enforcement scenarios: scenarios MP1 and MP2 capture the two different enforcement strategies in case of a missed pill, while scenario LP refers to a pill taken after the expected time.

Finally, to conclude stage 1 of the RSE process, we have defined in Fig. 8 the I/O and probing/effecting interfaces (detailed in the next section) of the pillbox using a UML-like notation.

6.2. MVM safety enforcement scenarios

MVM is a critical medical system. We have applied both black-box and gray-box enforcement.

6.2.1. Black-box safety enforcement scenarios

We have applied the black-box enforcement scenarios to check if the user inputs (e.g., respiratory rate, inspiratory pressure and apnea lag, etc.) are within their upper and lower limits. Here we show those relative to the timers set during the ventilation (Table 3):

- APNEA LAG WITHIN LIMITS: the apnea lag must be greater than or equal to 10 s and less than or equal to 60 s;

- I:E ratio : when in PCV mode, the ratio of inspiratory time to expiratory time must be in the interval 1:1–1:4.

In case safety assertions are violated, the enforcer activates the *fail-safe* configuration: the MVM is locked until the parameters are set by the user within their limits.

6.2.2. Gray-box safety enforcement scenarios

We report here two of the gray-box safety assertions that must be satisfied to ensure its safe operation:

- PATIENT STOPS BREATHING AUTONOMOUSLY: the patient must always breathe even if he or she is not able to start breathing on his/her own;
- MINIMUM WORK OF BREATHING: the muscular work to inflate or deflate the lungs must be minimal.

If the safety assertions are violated, we assume the enforcer automatically applies the following *safety enforcement strategies*:

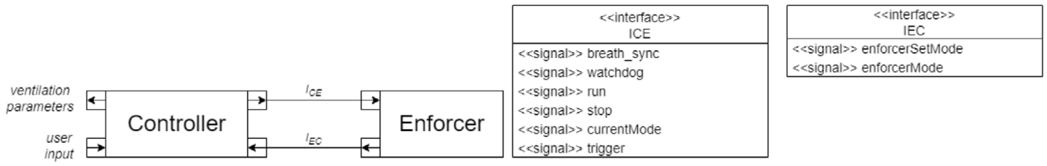

Fig. 9. MVM system and signals exchanged.

APNEA (AP): when in PSV mode, if the patient is not able to start a new breath within apnea lag, the enforcer forces the ventilator in PCV mode.

LUNG STRESS (LS): when in PCV mode, if the patient automatically trigger spontaneously three consecutive respiratory cycles, the enforcer forces the ventilator in PSV mode.

The two safety enforcement scenarios are reported in Table 4: scenario AP captures the enforcement strategy in case patient stops breathing autonomously, while scenario LS refers to guarantee the minimum work of breathing. A further unsafe situation not reported in Table 4 is when due to probe failing (signals are not received) the system is put into *safe-mode*. Fig. 9 shows the I/O and probing/effecting interfaces (detailed in the next section) of the MVM system using a UML-like notation.

7. Runtime enforcement models

This section illustrates excerpts of the enforcement models for the MRM and the MVM case studies (stage 2 of the RSE process).

7.1. MRM enforcement models

The instantiation of the enforcement framework for the MRM case study has been extensively presented in paper (Bonfanti et al., 2021); we summarize here only the most important information. The I/O interface of the pillbox with the user consists of the `openSwitch` input signal to open/close the compartment containing the pills, `outMess` and `redLed` (one for each compartment) to respectively show messages and status of compartments to the user.

Black-box enforcement model. An example of enforcement model to apply input sanitization strategies (black-box enforcement) is shown in Code 2, where the ASM module `PillboxSanitizer` is shown. It consists of a sequence of invariants, each corresponding to an enforcement scenario shown in Table 1: invariant `inv_PB1` allows checking the consistency between the prescription type and the compartments of the pillbox, invariant `inv_PB2` allows checking the amount consistency between the pills to intake and the number of the corresponding compartment slots, invariant `inv_PB3` allows checking the consistency of the time schedule of the compartment's slots and those scheduled for the medicine and it checks that no interferences are present, and invariant `inv_PB4` allows checking if there is inference between consecutive pills.

```
module pillbox_sanitiser
...
invariant inv_PB1 over medicine_list: (forall $c in Compartment with (
        contains(medicine_list,name($c))))
invariant inv_PB2 over Compartment: (forall $m in asSet(medicine_list) with (
        exist $c in Compartment with name($c)=$m))
invariant inv_PB3 over Compartment: (forall $c in Compartment with iton(
        length(time_consumption($c))) = amount(name($c)))
invariant inv_PB4 over Compartment, Medicine: state=INIT implies (forall $c
        in Compartment with (((drugIndex($c) < iton(length(time_consumption(
        $c))) and at(time_consumption($c),drugIndex($c)) = (at(time(name($c)),
        drugIndex($c)))))))
```

Code 2: PillboxSanitizer ASM model

Table 4
Gray-box safety enforcement scenarios MVM.

Scenario	Unsafe condition	Enforcement strategy	Enforcement plan
AP	Patient in Apnea	From PSV to PCV	Move to PCV
LS	Lung stress	From PCV to PSV	Move to PSV

Gray-box enforcement model. The enforcer monitors the pillbox through the following set of probes:

- `time_consumption`: instant of time in which the medicine must be taken;
- `redLed`: status (on/off/blinking) of the red led for each compartment;
- `drugIndex`: index of the medicine that must be taken;
- `actual_time_consumption`: time in which the medicine has been taken;
- `isPillMissed`: true if current pill has been missed;
- `pillTakenWithDelay`: true if `actual_time_ consumption` is different from `time_consumption`.

In case safety assertions are violated, the enforcer affects the monitored system by setting the following effectors:

- `setNewTime`: it indicates if the pill time consumption must be rescheduled;
- `newTime`: it sets the new time consumption;
- `setOriginalTime`: it indicates if the pill cannot be rescheduled;
- `skipNextPill`: it indicates if the pill must be skipped.

Code 3 reports the enforcement rule for the scenario LP when the pill is taken with delay (`pillTakenWithDelay`) and the enforcer verifies any overlaps with next pills.

As explained in Section 4.2, the enforcement rule has the form of an ECA rule. The unsafe condition $\neg\alpha$ happens when the guard `pillTakenWithDelay`, which is true on a pill if its `actual_time_consumption` is different from pill's `time_consumption`, holds, violating the safety consumption PILL ON TIME (the boolean value of `pillTakenWithDelay` is provided as probe event). The safety enforceability condition ρ is `delayCausesOverlap`, which is checked individually for all the next pills. It holds for each next pill if the time distance between the prescription time $t_c(nextPill)$ of the next pill and the actual consumption time $t_a(currentPill)$ of the taken pill is less than the minimum separation time $minToInterferer(currentPill, nextPill)$ between current pill and next pill, i.e.:

$$(t_c(nextPill) - t_a(currentPill)) < minToInterfer(currentPill, nextPill)$$

```
rule r_noOverlapping($compartment in Compartment)=
    if pillTakenWithDelay($compartment) then
      forall $c2 in next($compartment) do
        if delayCauseOverlap($compartment,$c2) then
          skipNextPill($compartment, $c2):= true endif endif
```

Code 3: Enforcement rule for scenario LP

The enforcement rule `r_pillOnTime` for scenarios MP1 and MP2 is already presented in Bonfanti et al. (2021).

7.2. MVM enforcement models

In this section, we introduce the enforcement strategies for the MVM case study. The input signals are all the user parameters required for the ventilation, e.g., respiratory rate, inspiratory pressure, apnea lag, etc. The output to the user are the ventilation parameters, e.g., airway pressure and tidal volume.

Black-box enforcement model. Code 4 shows the ASM enforcement model to apply input sanitization. It allows checking if the apnea lag is in the range 10–60 s (invariant `inv_APRange`) and I:E ratio is in the interval 1:1–1:4 (invariant `inv_IE`).

```
module pillbox_sanitiser
...
invariant inv_APRange over timerApneaLag: (duration(timerApneaLag)>=10 and
        duration(timerApneaLag)<=60)

invariant inv_IE over timerInspirationDurPCV,timerExpirationDurPCV: ((duration
       (timerExpirationDurPCV)>=duration(timerInspirationDurPCV)) and (
       duration(timerExpirationDurPCV)<=4*duration(timerInspirationDurPCV)))
```

Code 4: MVM model for input sanitization

Gray-box enforcement model. The enforcer monitors the ventilation through the following probes:

- `breath_sync`: respiration phase, inspiration or expiration;
- `watchdog`: set of bits reporting the controller status;
- `run`: ventilation starts;
- `stop`: ventilation stops;
- `currentMode`: current ventilation mode;
- `trigger`: in PCV is true if the transition from inspiration to expiration or vice-versa happens because of trigger events and not because of timers deadline.

Whenever safety assertions are violated, the enforcer influences the managed system by setting the following effectors:

- `enforcerSetMode`: it indicates if the enforcer forces new ventilation mode;
- `enforcerMode`: the mode set by the enforcer.

As an example, Code 5 reports the enforcement rule for scenario AP. The rule fires if the ventilation is in expiration (*breath_sync = EXP*) and the ventilation mode is PSV (*currentMode = PSV*). The unsafe condition holds when the patient is in apnea (`patientInApnea()`), i.e., when the expiration duration is greater than the apnea lag (*calc_t(timer_exp) > duration (timerApneaLag)*). Due to this high critical safety condition, which if not satisfied the patient could die, the safety enforceability condition ρ is always true, this means that if the unsafe condition happens, the enforcer always executes the enforcement plan, i.e. it forces the ventilation to PCV mode.

```
//Patient is in expiration and mode is PSV
rule r_apneaLag=
    if patientInApnea() then
        par
            enforcerMode := PCV
            enforcerSetMode := true
        endpar
    endif endif
```

Code 5: Enforcement rule for scenario AP

8. RSE framework implementation and instantiation

The RSE approach has been realized as a component framework and instantiated for the MRM and the MVM case studies (stage 3 of the RSE process). The software enforcer is designed to act as a proxy which wraps around the managed system. Although its code may be partially available in the form of a component framework for some core functionalities and the code of the enforcer logic be synthesized automatically by enforcement models (e.g., input/output automata used to specify the particular enforcement strategy), normally the software enforcer is *system-specific* and therefore some manual steps are required from the developers to implement those components (such as components for the causal connection to the managed system) that are abstracted in the enforcement model/framework and that are specific to the target system's domain.

In this respect, we developed a Java-based *white-box framework*[4] for common core functionalities that can be conveniently reused by concrete realization of sets of components to accomplish enforcement tasks for different systems. By exploiting the design patterns *Template* and *Factory*, the concrete components have an inheritance relationship with the framework's components[5]. We also combined the MAPE-K control loop style with the *Safety Assertion Enforcer* pattern (Fernandez and Hamid, 2017) for shaping the enforcer subsystem.

Fig. 10 shows a high-level overview of the enforcement framework architecture. The current version of RSE supports shell-like software applications as managed systems, and local pipes and ZeroMQ as communication means to connect the enforcer software with them. The component `I/O sanitizer` is responsible for exchanging I/O values with the `Managed System` and realizing black-box enforcement. The `I/O sanitizer` component exploits a runtime ASM as black-box enforcement model and therefore invokes (through the interface *ModelExecution*) the simulation service of the model engine `ASM@run.time Simulator`.

In the gray-box approach, the `Enforcer` subsystem adopts the MAPE-K control structure and the `Monitor` component exploits a pull-based interaction to regularly request new data to the probes of the managed system; the `Analyzer` starts only when the probes values change. A MAPE-K control loop is therefore regularly executed over time to assure the system safety requirements at runtime. The rule-based decision making of the `Analyzer` and `Planner` components is supported by the `ASM@run.time Simulator` that can run simulations of the ASM enforcement model during operation. The UML sequence diagram in Fig. 11 details the gray-box safety enforcement executed by the `Enforcer` operation *runFeedBackLoop()*, and, therefore, the MAPE-K components interactions to enact the appropriate enforcement strategy as suggested by the ASM model run by the `ASM@run.time Simulator`. Note that to avoid deadlock/huge delay in the system execution, the framework supports an execution mode with a configurable timeout, according to which the enforcer does not wait indefinitely for the simulator to finish the ASM run. Specifically, if the ASM model (for some reason) runs too long, its execution times out (according to a specific configurable time limit) and control returns without applying any enforcement strategy.

When the `I/O sanitizer` component is used in combination with a gray-box enforcement mechanism, it simply plays the role of a *delegator* that saves the I/O values into the `knowledge` repository and then delegates to the MAPE-K feedback loop the further

[4] RSE is available within the ASMETA GitHub repository https://github.com/asmeta/asmeta/tree/master/code/experimental/asmeta.enforcer .

[5] The concrete component classes implement the basic abstract methods and override the hook methods of the framework's abstract classes to add specific behaviors.

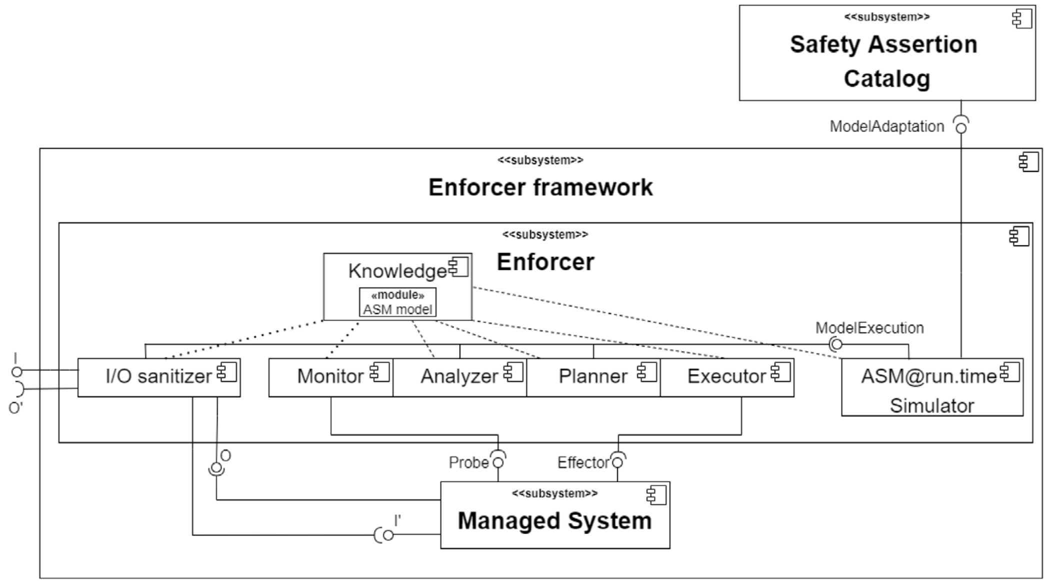

Fig. 10. Enforcement framework architecture.

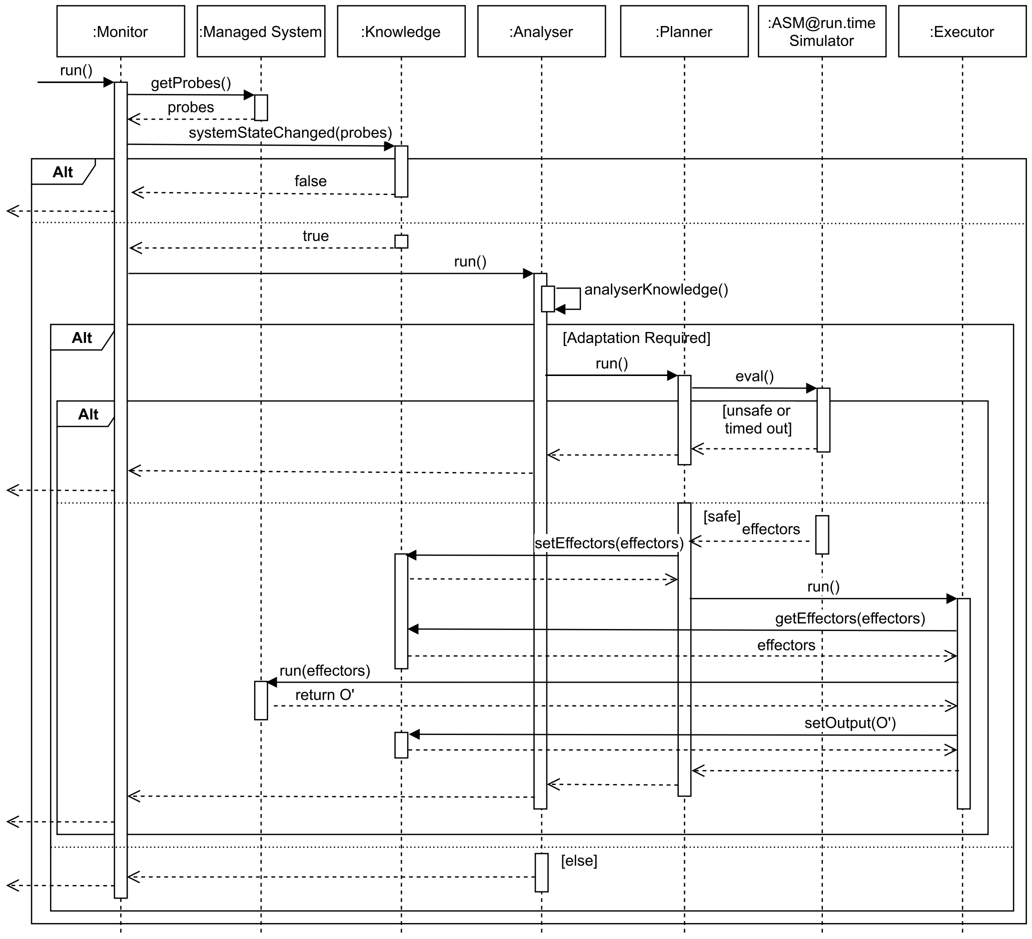

Fig. 11. Enforcer's operation *runFeedBackLoop()*.

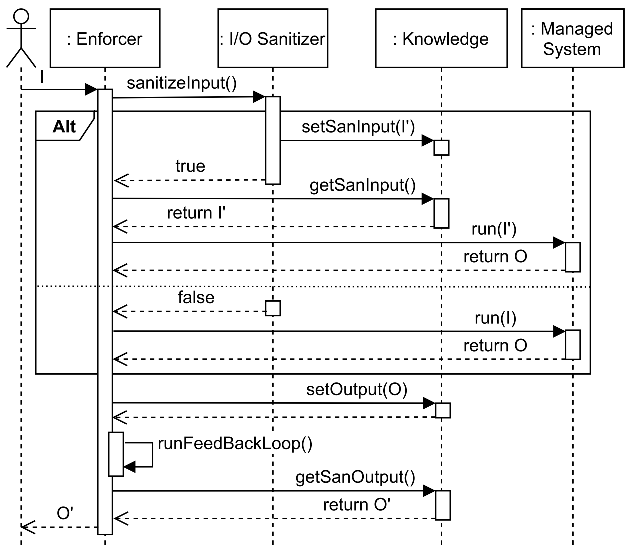

Fig. 12. Enforcement control flow.

gray-box enforcement actions. The UML sequence diagram in Fig. 12 shows the overall flow of iterations between the framework's components during an input sanitization followed by an output sanitization with adaptation executed by the `Enforcer` operation *runFeedBackLoop()*.

The subsystem `Safety Assertion Catalog` realizes a sort of dashboard for *Human-Model-Interplay* (both in a graphical and in a command-line way) to visualize the current status of execution and to enact commands for changing safety properties at model level. For this purpose, it consumes the interface `ModelAdaptation` provided by the `ASM@run.time Simulator`. In order to make on-the-fly changes of the underlying ASM runtime model consistently, a separate thread of the simulator (different from the model simulation thread) manages the model adaptation only when the status of the simulation reaches a *quiescent state*, i.e. no enforcement activity is going on. Then, the ASM model continues to execute from its current state. Adding a safety invariant that would be immediately violated in the current state of the ASM model is forbidden at the level of the user interface.

9. RSE evaluation

In this section, we evaluate our RSE approach in terms of precise research questions and discuss some threats to the RSE validity.

9.1. RSE at work

We here show the execution of the framework (stage 4 of the process) and validate it by means of the following two research questions: (RQ1) check that the RSE framework delivers the intended behavior in enforcing the safety assertions, and (RQ2) assess the computation overhead of the enforcement process on the overall system execution time.

Regarding RQ1, intuitively, we want to show that the enforcer modifies the system behavior according to the expected strategy and leaves the system behavior unaltered when the observed computation does not require any intervention. In fact, although the scope of the enforcer is to steer the behavior of the running software to stay in a safe region, the enforcer behavior itself may be not correct and compromise the running system behavior rather than enforcing it.

Regarding RQ2, we intent to evaluate the time overhead of running the enforcer in combination with the system.

In the next subsections we discuss, on two case studies, the results of our validation approach. For both the two case studies, the experiments were performed on Intel(R) Core(TM) i7-10700F CPU @ 2.90 GHz and 64 GB RAM.

(RQ1) Sound operation of the enforcement software. To answer this question, we evaluated the capability of the enforcer of producing an intended or expected outcome. As example, we report here four scenarios: one black-box and one gray-box for both MRM and MVM case studies.

The first scenario presented simulates the scenario PB3 presented in Table 1 (whose invariant is shown in Code 2): the time scheduled for the pill is not the same as prescribed by the doctor. The simulation trace is shown in Code 6: the simulation is immediately interrupted due to invariant violation (line 7): the time of the prescribed pill (`time_consumption(compartment1)[360]`) is different from the time set in the pillbox (`time_consumption (compartment1) [300]`). Note that the time is simulated by setting `systemTime`, which is defined in minutes from midnight instead of hours:minutes.

```
2  Pillbox initialization:
3  Output to pillbox: {}
4  User Input: {systemTime=361, openSwitch(compartment1)=false, openSwitch(
       compartment2)=false}
5  No transition to step 1 for model safePillbox.asm
6  Model execution outcome: UNSAFE
7  Reason: Invalid Invariant inv_PB3
```

Code 6: Simulation trace PB3 of MRM

In Code 7, we report an example of simulation trace for the LP scenario of the MRM case study (gray-box enforcement strategy). In this scenario the fosamax should be taken at 6:00 a.m. (`systemTime=360`), but the patient takes it at 6:16 a.m. (`systemTime=376`). The delay causes a safety property violation of the minimum time between fosamax and the next pill, moment is skipped.

When it is time to take fosamax (the first pill prescribed), the pill box turns on the corresponding red led and displays a message to the patient to take fosamax (line 5). After 10 min the user opens the compartment of the medicine (`openSwitch=true`), the red led starts blinking and the pill box reminds to the patient to close the compartment (line 12 — we assume that the pill is taken when the user closes the compartment). At 6:16 a.m. the user takes the pill (the user closes the compartment — line 19), but this causes the safety property violation and the enforcer identifies that the pill next to that in `compartment1` must be skipped (`skipNextPill(compartment1)=true`). The next pill in `compartment2`, namely moment (`skipNextPill(compartment1,compartment2)=true`) — line 23,[6] is skipped in order to steer the system into the safety region. Until this moment, the enforcer did not intervene. The message `SAFE` (e.g. in line 1) is the output of the *ASM@run.time* simulator through the `ModelExecution` interface; it states that the run time safety assurance in the `SafePillbox` model has been addressed.

[6] In ASMs, function overloading is allowed. In our model, the boolean-valued function symbol `skipNextPill` is defined twice: with one argument x to specify whether or not the pill in compartment x causes a skip of next pill; with two arguments x and y to specify whether or not the next pill in compartment y must be skipped.

```
1   Enforcer initialization... Model execution outcome: SAFE
2   Output to pillbox: {}
3   User Input: {systemTime=361, openSwitch(compartment1)=false, openSwitch(
        compartment2)=false}
4   Pillbox running...
5   Output to patient: {redLed(compartment2)=OFF, redLed(compartment1)=ON,
        outMess(compartment1)="Take fosamax"}
6   Output for probing: {redLed(compartment2)=OFF, time_consumption(
        compartment1)=[360], time_consumption(compartment2)=[730,1140],
        redLed(compartment1)=ON, drugIndex(compartment1)=0, isPillMissed(
        compartment1)=false, pillTakenWithDelay(compartment2)=false,
        systemTime=361, pillTakenWithDelay(compartment1)=false, drugIndex(
        compartment2)=0, name(compartment2)="moment", name(compartment1
        )="fosamax", isPillMissed(compartment2)=false, day=0}

8   Enforcer running... Model execution outcome: SAFE
9   Output to pillbox: {}
10  User Input: {systemTime=371, openSwitch(compartment1)=true, openSwitch(
        compartment2)=false}
11  Pillbox running...
12  Output to patient: {redLed(compartment2)=OFF, redLed(compartment1)=
        BLINKING, outMess(compartment1)="Close fosamax in 10 minutes"}
13  Output for probing: {redLed(compartment2)=OFF, time_consumption(
        compartment1)=[360], time_consumption(compartment2)=[730,1140],
        redLed(compartment1)=BLINKING, drugIndex(compartment1)=0,
        isPillMissed(compartment1)=false, pillTakenWithDelay(compartment2)=
        false, systemTime=371, pillTakenWithDelay(compartment1)=false,
        drugIndex(compartment2)=0, name(compartment2)="moment", name(
        compartment1)="fosamax", isPillMissed(compartment2)=false, day=0}

15  Enforcer running... Model execution outcome: SAFE
16  Output to pillbox: {}
17  User Input: {systemTime=376, openSwitch(compartment1)=false, openSwitch(
        compartment2)=false}
18  Pillbox running...
19  Output to patient: {redLed(compartment2)=OFF, redLed(compartment1)=OFF,
        outMess(compartment1)="fosamax taken"}
20  Output for probing: {redLed(compartment2)=OFF, time_consumption(
        compartment1)=[360], time_consumption(compartment2)=[730,1140],
        redLed(compartment1)=OFF, drugIndex(compartment1)=0, isPillMissed(
        compartment1)=false, pillTakenWithDelay(compartment2)=false,
        actual_time_consumption(compartment1)=[376], systemTime=376,
        pillTakenWithDelay(compartment1)=true, drugIndex(compartment2)=0,
        name(compartment2)="moment", name(compartment1)="fosamax",
        isPillMissed(compartment2)=false, day=0}

22  Enforcer running... Model execution outcome: SAFE
23  Output to pillbox: {skipNextPill(compartment1,compartment2)=true,
        skipNextPill(compartment1)=true}
24  User Input: {skipNextPill(compartment1)=true, systemTime=376, openSwitch(
        compartment1)=false, openSwitch(compartment2)=false, skipNextPill(
        compartment1,compartment2)=true}
25  Pillbox running...
26  Output to patient: {redLed(compartment2)=OFF, redLed(compartment1)=OFF,
        outMess(compartment1)="moment
        skipped"}
27  Output for probing: {redLed(compartment2)=OFF, time_consumption(
        compartment1)=[360], time_consumption(compartment2)=[730,1140],
        redLed(compartment1)=OFF, drugIndex(compartment1)=0, isPillMissed(
        compartment1)=false, pillTakenWithDelay(compartment2)=false,
        actual_time_consumption(compartment1)=[376], systemTime=376,
        pillTakenWithDelay(compartment1)=false, drugIndex(compartment2)=1,
        name(compartment2)="moment", name(compartment1)="fosamax",
        isPillMissed(compartment2)=false, day=0}
```

Code 7: Simulation trace LP of MRM

Code 8 shows the simulation trace for APRange scenario (black-box enforcement scenario). The user inputs are checked and the simulation is immediately interrupted due to invariant violation (line 6): the apnea lag is higher than 60 s (see invariant in Code 4).

```
1   MVMController initialization:
2   Output to MVM: {}
3   Input: {}
4   No transition to step 1 for model MVMController.asm
5   Model execution outcome: UNSAFE
6   Reason: Invalid Invariant inv_APRange
```

Code 8: Simulation trace APRange of MVM

Code 9 shows the simulation of the AP scenario of the MVM case study (gray-box enforcement strategy). For clarity, we report only the main parameters. After startup (line 5), self-test (line 12) and ventilation off (line 19) phases, the ventilation starts in PSV mode (line 24). The patient is inhaling, the input valve is open and the output valve is closed. Then, the patient starts expiration spontaneously (line 31). After apnea lag, the patient is not able to start a new breath, the `breath_sync` signal still equals to `EXP` (line 43). The enforcer intervenes and forces the ventilation in PCV mode allowing the patient to breath (line 46) controlled by the ventilator.

We have reported here three scenarios, one for the black-box enforcement strategy and two for the gray-box enforcement strategy. For all the scenarios, we have forced the system on possible unsafe situations, and we have observed that the enforcement system intervenes to enforce the system into safe situation (gray-box enforcement) or block the system to avoid possible future unsafe situation (black-box enforcement).

```
1   Enforcer initialization... Model execution outcome: SAFE
2   Output to MVM: {}
3   Input: {}
4   MVM running...
5   Output to user: {state=STARTUP}
6   Output for probing: {watchdog=INACTIVE}

8   Enforcer running... Model execution outcome: SAFE
9   Output to MVM: {}
10  Input: {startupEnded=true}
11  MVM running...
12  Output to user: {state=SELFTEST}
13  Output for probing: {watchdog=INACTIVE}

15  Enforcer running... Model execution outcome: SAFE
16  Output to MVM: {}
17  Input: {selfTestPassed=true}
18  MVM running...
19  Output to user: {state=VENTILATIONOFF}
20  Output for probing: {watchdog=INACTIVE}

22  Enforcer running... Model execution outcome: SAFE
23  Output to MVM: {}
24  Input: {startVentilation=true, respirationMode=PSV}
25  MVM running...
26  Output to user: {oValve=CLOSED, iValve=OPEN, state=PSV_STATE phase=
        INSPIRATION}
27  Output for probing: {watchdog=BREATHON, breath_sync=INSP, run=true,
        currentMode=PSV}

29  Enforcer running... Model execution outcome: SAFE
30  Output to MVM: {}
31  Input: {pawGTMaxPinsp=true}
32  MVM running...
33  Output to user: {oValve=OPEN, iValve=CLOSED, state=PSV_STATE, phase=
        EXPIRATION}
34  Output for probing: {watchdog=BREATHON, breath_sync=EXP, currentMode=PSV}

36  ... //after apnea lag

38  Enforcer running... Model execution outcome: SAFE
39  Output to MVM: {}
40  Input: {}
41  MVM running...
42  Output to user: {oValve=OPEN, iValve=CLOSED, state=PSV_STATE, phase=
        EXPIRATION}
43  Output for probing: {watchdog=BREATHON, breath_sync=EXP,
        currentMode=PSV}

45  Enforcer running... Model execution outcome: SAFE
46  Output to MVM: {enforcerSetMode=true, enforcerMode=PCV}
47  MVM running...
48  Output to user: {oValve=CLOSED, iValve=OPEN, state=PCV_STATE, phase=
        INSPIRATION}
49  Output for probing: {watchdog=BREATHON, breath_sync=INSP,
        currentMode=PCV}
```

Code 9: Simulation trace AP of MVM

(RQ2) Computation overhead of the enforcement software. To evaluate the computation overhead, we clocked the time required by the RSE framework only in the (most complex) gray-box mechanism to run the feedback loop and make enforcement decisions

Table 5
MRM execution time.

Scenario			Run 1	Run 2	Run 3	Run 4	Run 5
MP1	Pillbox system		47 ms 34%	20 ms 22%	9 ms 8%	12 ms	
	Enforcer		91 ms 66%	73 ms 78%	109 ms 92%		
		SafePillbox model	20 ms 14%	24 ms 26%	27 ms 23%		
	Total Time		138 ms	93 ms	118 ms		
MP2	Pillbox system		42 ms 34%	23 ms 23%	14 ms 9%	16 ms	
	Enforcer		83 ms 66%	75 ms 77%	134 ms 91%		
		SafePillbox model	20 ms 16%	32 ms 33%	36 ms 24%		
	Total Time		124 ms	98 ms	148 ms		
LP	Pillbox system		56 ms 42%	23 ms 19%	16 ms 15%	13 ms 10%	23 ms
	Enforcer		76 ms 58%	97 ms 81%	91 ms 85%	111 ms 90%	
		SafePillbox model	23 ms 17%	27 ms 23%	28 ms 26%	30 ms 24%	
	Total Time		132 ms	120 ms	107 ms	124 ms	

Table 6
MVM execution time.

Scenario			Run 1	Run 2	Run 3	Run 4	Run 5	Run 6
AP	MVM system		38 ms 30%	17 ms 25%	12 ms 27%	19 ms 23%	14 ms 10%	11 ms
	Enforcer		87 ms 70%	51 ms 75%	32 ms 73%	63 ms 77%	126 ms 90%	
		SafeMVM model	20 ms 16%	8 ms 12%	11 ms 25%	21 ms 26%	32 ms 23%	
	Total Time		125 ms	68 ms	44 ms	82 ms	140 ms	
LS	MVM system		17 ms 23%	12 ms 18%	18 ms 19%	22 ms 21%	15 ms 7%	14 ms
	Enforcer		58 ms 77%	54 ms 82%	75 ms 81%	83 ms 79%	187 ms 93%	
		SafeMVM model	9 ms 12%	13 ms 20%	21 ms 23%	25 ms 24%	51 ms 25%	
	Total Time		75 ms	66 ms	93 ms	105 ms	202 ms	

on the considered scenarios. We report the results in Tables 5 and 6 respectively for MRM and MVM case studies. For each scenario, we have collected the time required to run the system and to run the enforcer (including the run of the enforcement model). We have highlighted in blue the runs where the enforcer intervenes to avoid the violation of safety assertions by the system. The results show that on average, it takes approximately the 90% of total running time for determining and executing an enforcement plan when safety properties are violated. This overhead is mainly related to the execution time of the ASM model. Indeed, the `ASM@run.time Simulator` is based on the design-time simulator `AsmetaS`, which is a simulator of formal models, developed in Java for model validation purposes and not for obtaining fast performance. To overcome such limitation we should consider in the future the availability of an optimized version of the simulator that exploits specific optimizations and customized properties at the level of the Java virtual machine. The observed overhead is however acceptable for the class of systems where response time are relatively long, e.g., the MRM system where if a pill is taken few seconds later is not critical. In case of reactive systems, like MVM, where response time must be as short as possible, this solution should be optimized (see Section 9.2 for further discussion).

9.2. Threats to validity

Some potential threats to the validity of the proposed approach and its validation have been addressed as follows.

Construct validity threats may arise due to assumptions made when modeling safety assertions and enforcement strategies for the systems. To mitigate these threats, we used safety requirements based on known case studies from the research literature to which the authors contributed to.

To reduce threats to *internal validity*, we designed a set of controlled validation experiments detailing the enforcement scenarios of interest for both the two case studies. In particular, our validation setting allows for direct access to: the oracle and enforcement models, the RSE instances for the case studies, and the managed systems. This direct manipulation has been fundamental to assess cause–effect relations between the system under scrutiny and the enforcer software, and therefore to show the correctness of the RSE approach. We also enable replication by making our implementation and the validation results publicly available.

External validity threats may exist if the characteristics of the systems of our case studies are not indicative of the characteristics of other systems or not significant from the safety perspective. We limited these threats by adopting two case studies from the medical domain, which are highly safety-critical. The application of our approach to additional case studies in other domains is part of our future work. Another external validity threat may arise if the enforcement (adaptation) latency is not tolerated by the system under scrutiny. The system needs sufficient time to react to the decisions of the enforcer and to change its behavior to avoid unsafe situations. The ideal case, in order to keep the model in a consistent and safe state, is that during the enforcement process the enforcer freezes the overall system computation. This is how the enforcer works in the MRM case study. However, this is a strong assumption which is not applicable to some systems like the MVM system. In this second case study, in fact, we had to decouple the enforcer and the system behaviors; when the enforcer is checking safety, the MVM continuously runs the ventilation process. In this second situation, if the enforcer spends too much time to generate the enforcement plan, the safety violation might occur. In order to mitigate this risk, our approach adopts quick and precisely pre-computed enforcement strategies encoded in terms of ECA rules, instead of generating them or adopting sophisticated reasoning and planning. In the specific case of the MVM system, the enforcement latency and the reliability of the communication mean could be also improved by synthesizing the enforcer software (in a re-engineering process) directly in the embedded software of the ventilator, possibly using an optimized version of the ASM simulator with a version of the JVM for embedded installations. However, in very high time-critical situations, which require quick and precisely specified reactions by the system, a different method should be adopted for the enforcement of timed safety assertions (such as the approach theoretically defined in Renard et al., 2020).

A further threat regarding external validity concerns the problem of addressing enforcement strategies (and therefore, rules) that may be in conflict. In our case studies, we do not have conflicting strategies, however, they can arise during model validation at design time and can be corrected (at design time) on the base of established priorities. The ASM enforcement rules are able to reflect possible priorities. The approach of resolving conflicting enforcement strategies at design time is similar to that presented in Arcaini et al. (2020, 2017a) for resolving conflicts among self-adaptation goals of decentralized MAPE loops. By exploiting the model review process (Arcaini et al., 2010) on ASM models, it is possible to discover conflicting state locations by checking precise meta-properties on the model. When conflicts arise, it is up to the designer to resolve them by specifying, in terms of new or redefined rules, possible cooperation and coordination of the enforcement actions to avoid interference and provide certain guarantees about enforcement. Conflicts discovered at run-time and not foreseen at design time are possible; they are not dealt in this paper since they do not apply to our case studies, but their resolution would require evolving the model into a new one, able to prevent the conflicts according to design-time decisions (as explained above). Currently, in case of run-time conflicts, our framework brings the systems into a fail-safe configuration. As mentioned in the conclusions, dialing with conflict situations is an argument for future work.

10. Related work

First, we summarize the representative approaches on runtime safety assurance and then report only works related to runtime enforcement.

10.1. On runtime assurance of system safety

Some runtime certification methodologies in the context of self-adaptive systems, such as, for example, ConSerts (Trapp and Schneider, 2014) and ENTRUST (Calinescu et al., 2018), have been proposed for the dynamic provision of assurance cases by transferring activities of the safety assurance process to the runtime and arguing the suitability of a software system for its intended application at operation time. In line with these approaches, other formal approaches that cover trustworthiness evaluation, run-time maintenance, and evidence-based assurance specifically for Cyber-Physical Systems have been proposed (see, for example, Schubert et al., 2016; Mohammadi, 2019; Reich et al., 2020; Wu et al., 2017).

While runtime formal verification approaches (such as those considered in the study Calinescu and Kikuchi, 2011, Arcaini et al., 2012, Liang et al., 2009, etc.) generally focus on the *oracle problem*, namely assigning verdicts to a system execution, mechanisms for runtime enforcement (like the one we propose) focus on ensuring the correctness of the sequence of events by possibly modifying or preventing the system execution (Falcone et al., 2018).

In general, runtime enforcement and self-healing techniques are two classes of relevant solutions to deal with failures at runtime; they are complementary, although related (Falcone et al., 2018). Runtime enforcement techniques can prevent misbehavior of a target monitored system by enforcing the system to run according to its specification, while healing techniques can react to misbehavior and failures to restore the normal execution of the monitored system after a failure has been observed. Of course, the boundaries between these two classes of techniques are not always well defined, and some approaches in one category may have characteristics in common with approaches in the other category (Falcone et al., 2018).

10.2. On runtime enforcement techniques

Enforcement mechanisms were initially proposed in the security domain to enforce security and privacy policies (Erlingsson and Schneider, 1999). Runtime enforcement was also applied to enforce usage-control policies on mobile devices (Riganelli et al., 2017, 2019): a policy enforcement framework based on the concept of *proactive library* is presented and applied to the Android platform using run-time hooking and code injection mechanisms to handle faults caused by bad resource management such as a mobile device. Though this approach to enforce adaptation policies is somehow related to our, it is to be considered also a self-healing solution (Falcone et al., 2018) for a resource-constrained environment. Indeed, instead of preventing misbehavior (like in our enforcement mechanism), it reacts to misbehavior after it occurs.

In the robotic application domain, some enforcer approaches have been proposed that exploit *reactive synthesis* (or *logical synthesis*) (Church, 1964) to automatically generate correct-by-construction enforcer modules from high-level formal specifications (e.g., formulas in linear temporal logic).

To assure system safety at runtime, for example, in Desai et al. (2019) a robotics programming framework, called SOTER, is presented for implementing and testing high-level reactive robotics software like drone surveillance systems. This framework integrates also a runtime assurance (RTA) system for the use of uncertified components while still providing safety guarantees. Each uncertified component is protected using a RTA module (based on an encoded state machine) that guarantees safety by switching to a safe controller in case of danger. A similar safety assurance technique at runtime in the same application domain is presented in de Niz et al. (2018) to guarantee the safe behavior of a drone controller. This runtime assurance strategy consists into adding small components, called enforcers, to a drone system to monitor and steer the movement commands of the drone that must be restricted to fly within a constrained area. These enforcers are small components fully specified and verified at design time. To enable the verification of the composition of multiple enforcers, the enforcers are specified as SMT formulae and executed with the Z3 verifier. Another similar approach based on correct-by-construction synthesis of a monitor, which is proven to enforce model compliance at runtime for 2D waypoint-following of Dubins-type ground robots, has been proposed in Bohrer et al. (2019) using differential dynamic logic (dL) to model a robot and its kinematics. Differently from our framework potentially adoptable in any domain, all these approaches (Desai et al., 2019, de Niz et al., 2018, and Bohrer et al., 2019) are specific to the robotic domain and work well only for enforcer modules to synthesize of small size due to the same limitations of the logical and algorithmic solutions of the synthesis problem.

In Wu et al. (2017), a general approach to automatically synthesize enforcer components, called *safety guards*, from a formal specification (a Mealy machine) of safety properties for safety-critical cyber–physical systems is presented. Other similar approaches based on the synthesis of enforcement mechanisms from automaton-based formal specifications of the enforcement strategy are reviewed in Falcone et al. (2018). Unlike all these synthesis approaches that have been validated on small benchmarking examples and can be, as any code generation, error-prone and difficult to test, we rely instead on using directly the enforcement formal model at run-time by connecting it directly with the target system, namely we carry out *run-time execution (or simulation)* of the safety formal specification in tandem with the target system. The software enforcer relies on a feedback loop equipped with an enforcement model (an ASM@run.time) that is updated at run-time, when new knowledge about safety changing conditions becomes available; the enforcer uses this model to plan and apply an enforcement strategy (according to the behavior specified in the ASM@run.time) by adapting the target system. The proposed approach was inspired from model-based simulation at runtime successfully adopted to enable efficient decision-making in self-adaptive systems, usually to assess extra-functional quality properties (Weyns and Iftikhar, 2016). Simulation, in general, is less time and resource consuming compared with exhaustive verification techniques, and it is, therefore, particularly advantageous at run-time, when time and resources are often constrained (Calinescu and Kikuchi, 2011).

11. Conclusion

We proposed a runtime safety enforcement framework to assure safe execution of a software system. The framework monitors and keeps the system behavior in line with a set of safety requirements by anticipating incorrect behavior and countering it before it actually happens. The enforcer acts as an autonomic manager that wraps around the managed system and analyses and controls the system's operational changes and interactions with the environment with the help of an ASM used as runtime enforcement model.

In general, formal methods and analysis techniques are fundamental for ensuring the safe and correct behavior of a system and for mitigating potential hazards. The two trends, namely design-time and run-time analyses, are pointing to a new way of analyzing safety-critical software, which embraces the rigor of safety-critical formal analysis environments during the system design or development, while experiencing the benefits of run-time analysis when the system is operating.

In the future, we want to evaluate the generality of the proposed enforcement framework by targeting other systems and real-world application domains. In particular, we want to adopt our enforcement framework as an *autonomic middleware* to retrofit pre-existing systems/legacy systems (e.g., *Software as a Medical Device*) and secure them with external autonomic capabilities, thus avoiding costly device offline repairs/recalls. Furthermore, we will investigate on the problem of deciding whether a given set of (possible interfering) safety goals are enforceable or not through a gray-box enforcement mechanism. To this end we want to explore the use of ASMs for formally specifying complex enforcement strategies and for automatically detecting possible interferences. For this refined setting, we will give necessary and sufficient conditions on when and how a safety assertion is enforceable during the execution of the system and of the MAPE-K loop. In addition, we want to explore the applicability of our safety enforcement approach to autonomous systems whose runtime behavior is highly unpredictable, such as AI-based systems. In particular, we want to explore the runtime use of probabilistic ASMs or other state-based stochastic formalism to model and incorporate uncertainty issues explicitly (Camilli et al., 2020) within the enforcement process.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We have shared the web link to data/code within the paper.

References

Abba, A., et al., 2021. The novel mechanical ventilator milano for the COVID-19 pandemic. Phys. Fluids 33 (3), 037122. http://dx.doi.org/10.1063/5.0044445.

Andersson, B., Chaki, S., de Niz, D., 2017. Combining symbolic runtime enforcers for cyber-physical systems. In: Lahiri, S.K., Reger, G. (Eds.), Runtime Verification - 17th International Conference, RV 2017. In: Lecture Notes in Computer Science, Vol. 10548, Springer, http://dx.doi.org/10.1007/978-3-319-67531-2_5.

Andersson, J., Grassi, V., Mirandola, R., Perez-Palacin, D., 2020. A conceptual framework for resilience: fundamental definitions, strategies and metrics. Computing http://dx.doi.org/10.1007/s00607-020-00874-x.

Anon, 2022. ASMETA (ASM mETAmodeling) toolset. URL https://asmeta.github.io/.

Arcaini, P., Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P., 2021. The ASMETA approach to safety assurance of software systems. In: Raschke, A., Riccobene, E., Schewe, K. (Eds.), Logic, Computation and Rigorous Methods - Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday. In: Lecture Notes in Computer Science, Vol. 12750, Springer, pp. 215–238.

Arcaini, P., Gargantini, A., Riccobene, E., 2010. Automatic review of abstract state machines by meta property verification. In: Muñoz, C.A. (Ed.), Second NASA Formal Methods Symposium - NFM 2010, Washington D.C., USA, April 13-15, 2010. Proceedings. In: NASA Conference Proceedings, NASA/CP-2010-216215, pp. 4–13.

Arcaini, P., Gargantini, A., Riccobene, E., 2012. CoMA: Conformance monitoring of Java programs by abstract state machines. In: Runtime Verification. In: Lecture Notes in Computer Science, Vol. 7186, Springer, http://dx.doi.org/10.1007/978-3-642-29860-8_17.

Arcaini, P., Mirandola, R., Riccobene, E., Scandurra, P., 2020. MSL: a pattern language for engineering self-adaptive systems. J. Syst. Softw. 164, 110558. http://dx.doi.org/10.1016/j.jss.2020.110558.

Arcaini, P., Riccobene, E., Scandurra, P., 2015. Modeling and analyzing MAPE-K feedback loops for self-adaptation. In: Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. ACM, http://dx.doi.org/10.1109/SEAMS.2015.10.

Arcaini, P., Riccobene, E., Scandurra, P., 2017a. Formal design and verification of self-adaptive systems with decentralized control. ACM Trans. Auton. Adapt. Syst. 11 (4), http://dx.doi.org/10.1145/3019598.

Arcaini, P., Riccobene, E., Scandurra, P., 2017a. Formal design and verification of self-adaptive systems with decentralized control. ACM Trans. Auton. Adapt. Syst. 11 (4), 25:1–25:35. http://dx.doi.org/10.1145/3019598.

Bennaceur, A., France, R., Tamburrelli, G., Vogel, T., Mosterman, P.J., Cazzola, W., Costa, F.M., Pierantonio, A., Tichy, M., Akşit, M., Emmanuelson, P., Gang, H., Georgantas, N., Redlich, D., 2014. Mechanisms for leveraging models at runtime in self-adaptive software. In: Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (Eds.), Models@Run.Time: Foundations, Applications, and Roadmaps. Springer International Publishing, Cham, pp. 19–46. http://dx.doi.org/10.1007/978-3-319-08915-7_2.

Bennaceur, A., Ghezzi, C., Tei, K., Kehrer, T., Weyns, D., Calinescu, R., Dustdar, S., Hu, Z., Honiden, S., Ishikawa, F., Jin, Z., Kramer, J., Litoiu, M., Loreti, M., Moreno, G., Müller, H., Nenzi, L., Nuseibeh, B., Pasquale, L., Reisig, W., Schmidt, H., Tsigkanos, C., Zhao, H., 2019. Modelling and analysing resilient cyber-physical systems. In: 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS, http://dx.doi.org/10.1109/SEAMS.2019.00018.

Bohrer, R., Tan, Y.K., Mitsch, S., Sogokon, A., Platzer, A., 2019. A formal safety net for waypoint-following in ground robots. IEEE Robot. Autom. Lett. 4 (3), 2910–2917. http://dx.doi.org/10.1109/LRA.2019.2923099.

Bombarda, A., Bonfanti, S., Gargantini, A., 2019. Developing medical devices from abstract state machines to embedded systems: A smart pill box case study. In: Mazzara, M., Bruel, J.-M., Meyer, B., Petrenko, A. (Eds.), Software Technology: Methods and Tools. Springer International Publishing, Cham.

Bonfanti, S., Riccobene, E., Scandurra, P., 2021. A runtime safety enforcement approach by monitoring and adaptation. In: Biffl, S., Navarro, E., Löwe, W., Sirjani, M., Mirandola, R., Weyns, D. (Eds.), Software Architecture. Springer International Publishing, Cham, pp. 20–36. http://dx.doi.org/10.1007/978-3-030-86044-8_2.

Börger, E., Raschke, A., 2018. Modeling Companion for Software Practitioners. Springer, Berlin, Heidelberg, http://dx.doi.org/10.1007/978-3-662-56641-1.

Calinescu, R., Kikuchi, S., 2011. Formal methods @ runtime. In: Calinescu, R., Jackson, E. (Eds.), Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems. Springer Berlin Heidelberg, http://dx.doi.org/10.1007/978-3-642-21292-5_7.

Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M.U., Habli, I., Kelly, T., 2018. Engineering trustworthy self-adaptive software with dynamic assurance cases. IEEE Trans. Softw. Eng. 44 (11), http://dx.doi.org/10.1109/TSE.2017.2738640.

Camilli, M., Gargantini, A., Scandurra, P., 2020. Model-based hypothesis testing of uncertain software systems. Softw. Test. Verification Reliab. 30 (2), http://dx.doi.org/10.1002/stvr.1730.

Church, A., 1964. Logic, arithmetic, and automata. J. Symbolic Logic 29 (4), 210. http://dx.doi.org/10.2307/2270398.

de Niz, D., Andersson, B., Moreno, G., 2018. Safety enforcement for the verification of autonomous systems. In: Dudzik, M.C., Ricklin, J.C. (Eds.), Autonomous Systems: Sensors, Vehicles, Security, and the Internet of Everything. Vol. 10643, SPIE, International Society for Optics and Photonics, http://dx.doi.org/10.1117/12.2307575.

Desai, A., Ghosh, S., Seshia, S.A., Shankar, N., Tiwari, A., 2019. SOTER: A runtime assurance framework for programming safe robotics systems. In: 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. http://dx.doi.org/10.1109/DSN.2019.00027.

Erlingsson, Ú., Schneider, F.B., 1999. SASI enforcement of security policies: A retrospective. In: Proceedings of the 1999 Workshop on New Security Paradigms. NSPW '99, Association for Computing Machinery, http://dx.doi.org/10.1145/335169.335201.

Falcone, Y., Mariani, L., Rollet, A., Saha, S., 2018. Runtime failure prevention and reaction. In: Bartocci, E., Falcone, Y. (Eds.), Lectures on Runtime Verification: Introductory and Advanced Topics. Springer International Publishing, Cham, http://dx.doi.org/10.1007/978-3-319-75632-5_4.

Falcone, Y., Mounier, L., Fernandez, J., Richier, J., 2011. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods Syst. Des. 38 (3), http://dx.doi.org/10.1007/s10703-011-0114-4.

Fernandez, E.B., Hamid, B., 2017. Two safety patterns: Safety assertion and safety assertion enforcer. In: Proceedings of the 22nd European Conference on Pattern Languages of Programs. EuroPLoP '17, Association for Computing Machinery, http://dx.doi.org/10.1145/3147704.3147737.

Garlan, D., Schmerl, B.R., Cheng, S., 2009. Software architecture-based self-adaptation. In: Autonomic Computing and Networking. pp. 31–55. http://dx.doi.org/10.1007/978-0-387-89828-5_2.

He, Y., Schumann, J., 2020. A framework for the analysis of adaptive systems using bayesian statistics. In: Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. http://dx.doi.org/10.1145/3387939.3391596.

Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Computer 36 (1), http://dx.doi.org/10.1109/MC.2003.1160055.

Kramer, J., 2020. RE @ runtime : the challenge of change RE'20 Conference Keynote. In: Breaux, T.D., Zisman, A., Fricker, S., Glinz, M. (Eds.), 28th IEEE International Requirements Engineering Conference, RE 2020, Zurich, Switzerland, August 31 - September 4, 2020. IEEE, pp. 4–6. http://dx.doi.org/10.1109/RE48521.2020.00012.

Leveson, N., 2020. Are you sure your software will not kill anyone? Commun. ACM 63 (2), http://dx.doi.org/10.1145/3376127.

Liang, H., Dong, J.S., Sun, J., Wong, W.E., 2009. Software monitoring through formal specification animation. ISSE 5 (4), http://dx.doi.org/10.1007/s11334-009-0096-1.

Lutz, R.R., 2000. Software engineering for safety: A roadmap. In: Proceedings of the Conference on the Future of Software Engineering. ICSE '00, Association for Computing Machinery, http://dx.doi.org/10.1145/336512.336556.

Mohammadi, N.G., 2019. Trustworthy Cyber-Physical Systems - a Systematic Framework Towards Design and Evaluation of Trust and Trustworthiness. Springer Vieweg, http://dx.doi.org/10.1007/978-3-658-27488-7.

Reich, J., Schneider, D., Sorokos, I., Papadopoulos, Y., Kelly, T., Wei, R., Armengaud, E., Kaypmaz, C., 2020. Engineering of runtime safety monitors for cyber-physical systems with digital dependability identities. In: Casimiro, A., Ortmeier, F., Bitsch, F., Ferreira, P. (Eds.), Computer Safety, Reliability, and Security - 39th International Conference, SAFECOMP 2020, Lisbon, Portugal, September 16-18, 2020, Proceedings. In: Lecture Notes in Computer Science, Vol. 12234, Springer, http://dx.doi.org/10.1007/978-3-030-54549-9_1.

Renard, M., Rollet, A., Falcone, Y., 2020. Runtime enforcement of timed properties using games. Formal Aspects Comput. 32 (2–3), 315–360. http://dx.doi.org/10.1007/s00165-020-00515-2.

Riccobene, E., Scandurra, P., 2014. A formal framework for service modeling and prototyping. Formal Aspects Comput. 26 (6), http://dx.doi.org/10.1007/s00165-013-0289-0.

Riccobene, E., Scandurra, P., 2020a. Exploring the concept of abstract state machines for system runtime enforcement. In: Raschke, A., Méry, D., Houdek, F. (Eds.), Rigorous State-Based Methods. Springer International Publishing, Cham, pp. 244–247. http://dx.doi.org/10.1007/978-3-030-48077-6_18.

Riccobene, E., Scandurra, P., 2020b. Model-based simulation at runtime with abstract state machines. In: Software Architecture - 14th European Conference, ECSA 2020 Tracks and Workshops, Proceedings. In: Communications in Computer and Information Science, Vol. 1269, Springer, http://dx.doi.org/10.1007/978-3-030-59155-7_29.

Riganelli, O., Micucci, D., Mariani, L., 2017. Policy enforcement with proactive libraries. In: 12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems. IEEE Computer Society, http://dx.doi.org/10.1109/SEAMS.2017.9.

Riganelli, O., Micucci, D., Mariani, L., 2019. Controlling interactions with libraries in android apps through runtime enforcement. ACM Trans. Auton. Adapt. Syst. 14 (2), http://dx.doi.org/10.1145/3368087.

Schubert, D., Heinzemann, C., Gerking, C., 2016. Towards safe execution of reconfigurations in cyber-physical systems. In: 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE 2016, Venice, Italy, April 5-8, 2016. IEEE Computer Society, http://dx.doi.org/10.1109/CBSE.2016.10.

Tendeloo, Y.V., Mierlo, S.V., Vangheluwe, H., 2019. A multi-paradigm modelling approach to live modelling. Softw. Syst. Model. 18 (5), http://dx.doi.org/10.1007/s10270-018-0700-7.

Trapp, M., Schneider, D., 2014. Safety assurance of open adaptive systems – a survey. In: Models@Run.Time: Foundations, Applications, and Roadmaps. Springer International Publishing, Cham, http://dx.doi.org/10.1007/978-3-319-08915-7_11.

Weyns, D., Iftikhar, M.U., 2016. Model-based simulation at runtime for self-adaptive systems. In: Kounev, S., Giese, H., Liu, J. (Eds.), 2016 IEEE International Conference on Autonomic Computing, ICAC 2016. IEEE Computer Society, http://dx.doi.org/10.1109/ICAC.2016.67.

Wu, M., Zeng, H., Wang, C., Yu, H., 2017. Safety guard: Runtime enforcement for safety-critical cyber-physical systems: Invited. In: Proceedings of the 54th Annual Design Automation Conference. ACM, http://dx.doi.org/10.1145/3061639.3072957.