Towards a More Sustainable Re-Engineering of Heterogeneous Distributed Systems Using Cooperating Run-Time Monitors

Maximilian Gehring[1][0009−0004−5599−5807] and
Heiko Mantel[1][0000−0002−6586−5529]

Computer Science Department, TU Darmstadt, Germany
{maximilian.gehring1,heiko.mantel}@tu-darmstadt.de

Abstract. We propose an approach to using run-time monitoring for the re-engineering of distributed systems. Our re-engineering method MBRE consists of five steps that provide guidance during a re-engineering. By utilizing run-time monitors that are parametric in a policy, we obtain the flexibility needed for a sustainable integration of monitors. We illustrate this and other features of MBRE at three re-engineering case studies in a hypothetical hospital scenario. A key novelty of our approach is that it supports cooperation between monitors also across technological boundaries. This is of high relevance because, for instance, system components in the IoT often run on different platforms and are implemented in different programming languages. Surprisingly, such scenarios have been outside the focus of the run-time-monitoring community so far.

1 Introduction

A run-time monitor supervises runs of a given *target*, which may be an entire system or a system component, with respect to a given policy. The purpose is to detect policy violations while the target is running and, possibly, to prevent such violations. Naturally, what a monitor can detect is limited by the information that the monitor can observe when the target performs a step. If a step of the target would cause a policy violation, then the monitor might be able to take countermeasures to avoid the violation. Again, what a monitor can enforce is limited by how the monitor can influence the behavior of its target.

The choice of a suitable system layer is a key step in the integration of run-time monitoring into a system. Run-time monitors can be implemented, for instance, on the application layer, as part of a virtual machine or the operating system, or as a hardware component. As a rule of thumb, the lower the system layer, the more fine-grained a monitor's observations can be, and the higher the system layer, the more information a monitor can have about the semantics of steps and data. There is no system layer that is most suitable for integrating run-time monitors, in general. The suitability of a system layer usually depends on both, the application and the properties to be controlled by run-time monitoring.

In IoT-applications, one usually faces a situation, where some components comprise more system layers, into which monitors could be integrated, than other

components. For instance, a controller might only offer a simple processing unit on which machine code runs directly without any operating system or protective virtual machine. Thus, this is the only layer into which a software implementation of monitors could be integrated. In contrast, a server in the back-end might comprise a much richer set of system layers, including, e.g., the operating system, virtual machines, and applications running within these virtual machines. Moreover, even if two components conceptually feature the same system layer, these layers might differ substantially on a technical level, e.g., because a different processor is used. Finally, the software running on different components of an IoT system might have been implemented using different languages.

In this article, we focus on the re-engineering of distributed IoT systems and report on an on-going research project. At this stage, our main contribution is a novel re-engineering method that builds on the integration of a run-time monitoring framework into a given legacy system. We obtain flexibility by using monitors that can be instantiated with different policies. We determine the placement of monitors, the capabilities of these monitors to influence their respective target's behavior, and the need for additional sensors and actuators by a careful analysis of the legacy system and of the re-engineering objectives.

Our method for run-time-monitoring-based re-engineering consists of five steps. In the first step, the legacy system and the requirements are analyzed from a global perspective. By a decomposition, in the second step, one obtains a local perspective that fits what can be controlled by decentralized, cooperating monitors. The third and fourth step concern the technical integration of run-time monitors and of additional devices, respectively. The fifth step finalizes the re-engineering by instantiating all monitors with suitable policies. For validating the results of intermediate steps, our method offers the option to instantiate monitors with auxiliary policies that provide a basis for such a validation.

We use a hypothetical hospital as a playground for illustrating our method. In this setting, we consider routine tasks that might interfere with each other due to resource conflicts. We illustrate how monitors can be utilized to resolve such conflicts on top of the resolution provided by already existing organizational policies. We also illustrate how monitors integrated during re-engineering can be re-used in subsequent changes. This provides first evidence that re-engineering a legacy system with our method leads to flexibility that is sustainable.

The body of this article is structured as follows. In Section 2, we introduce our MBRE Method, the hypothetical hospital, and seven use cases. In Section 3, we apply our re-engineering method in three case studies of increasing complexity. In Section 4, we discuss implementation issues and report on first findings regarding the performance overhead induced by run-time monitors. We discuss related work in Section 5 before concluding in Section 6.

2 Approach and Application Scenario

The re-engineering of IT-systems is typically motivated by concrete needs. The possible motivations include the desire for additional or better functionality, the requirement to satisfy new regulations concerning safety or security, and the need

for more cost-effective operation. In addition, a re-engineering can be triggered by technological changes, e.g., by the availability of a better hardware platform.

In practice, re-engineering can be challenging and costly. Moreover, it can be difficult to attract qualified engineers to re-engineering projects, in particular, if they have made painful experiences with re-engineering before. Hence, it is desirable to simplify later changes pro-actively when building a system by choosing a flexible design that eases later adaptations. Albeit proactive flexibility is desirable, flexibility often needs to be added as an afterthought in practice, and this is the scenario on that we focus in this article. Nevertheless, we strive with our approach for similar advantages as in a pro-active integration of flexibility.

We split a re-engineering into two phases, the integration of run-time monitors into a legacy system and the instantiation of these monitors with policies that are suitable for enforcing the given requirements. By using monitors that can be instantiated with different policies, we also create flexibility for future changes. Should additional requirements arise, any previously integrated monitors can be re-used while adapting their policies to the re-modified requirements.

In addition, we exploit the possibility to exchange policies for validating that the integration of monitors into a legacy system does not cause undesired side effects. To this end, we instantiate the run-time monitors with dummy policies and test the resulting system against the legacy system without monitors.

To illustrate our approach, we focus on a particular application domain, i.e., the logistics within hospitals. In this domain, many processes are carefully regulated, for safety reasons, and optimized, for cost control. Nevertheless, additional room for improvement might appear in daily operation.

We introduce a hypothetical hospital as our "legacy system" in Section 2.2.

2.1 The MBRE Method: Monitor-Based Re-Engineering

The starting point of a re-engineering with our method consists of a legacy system and of a set of requirements that shall be satisfied by a re-engineering:

- We assume that the legacy system is a distributed system consisting of multiple hardware devices and of code running on these devices. In our case studies, we consider a legacy system that is accompanied by a floor plan and by multiple use-cases describing routine tasks. If a more informative documentation is available, (e.g., a definition of the original requirements, a documentation of the system design, or a more general development documentation), then this could be exploited in the analysis steps of our method.
- In our case studies, we use incidents encountered in daily operation to motivate a re-engineering. More generally, requirements could include the addition of new functionality, the improvement of existing functionality, performance improvements, or the compliance with safety or security regulations.

Throughout this article, we use requirement definitions that leave little freedom in how the problems shall be resolved conceptually. To apply our re-engineering method in case of less concrete requirements, one needs to make design decisions for arriving at sufficiently concrete requirements in a pre-processing step. How

Starting Point		
Input:	distributed legacy system	
	documentation of this system (including a collection of use cases)	
	list of requirements to be satisfied	
Step 1: Global analysis of the legacy system and of the requirements		
Output:	list of components to be supervised by monitors	[from Step 1c]
	list of devices to be added to the system	[1d]
	collection of modified use cases	[1e]
Step 2: Decomposition into local requirements		
Output:	specification of local logic of each monitor	[2b]
	specification of communication between monitors	[2c]
	specification of communication of monitors with devices	[2d, 2e]
Step 3: Integration of the run-time monitoring framework		
Output:	mapping of targets to system layers and integration techniques	[3a]
	legacy system with monitors integrated	[3b,3c]
	optional: auxiliary policies	[3d]
Step 4: Integration of additional devices		
Output:	legacy system with monitors and additional devices integrated	[4c]
	optional: auxiliary policies	[4d]
Step 5: Definition of policies and instantiation of monitors		
Output:	modified legacy system	[5b,5c,5d]

Fig. 1. The five steps of the MBRE Method and their respective output

to arrive at such a concretized requirement definition is a relevant topic in its own right, but not the focus of this article. Therefore, we abstract from it.

Figure 1 provides an overview of the five steps of our re-engineering method and their respective output. Steps 1 and 2 primarily operate on a conceptual level, while Steps 3 and 4 primarily focus on a technical level. The fifth step concerns both levels. In the following, we refine each of these five steps into finer-grained steps and introduce labels for the fine-grained steps. We use these labels in Figure 1 to clarify at which point each output is generated.

Global Analysis [Step 1]: Firstly, one identifies which use cases in the documentation of the legacy system must be adapted to satisfy the requirements [1a]. Secondly, one identifies which capabilities to obtain information at run-time are needed for enforcing the requirements and, in addition, which capabilities to alter the legacy system's behavior at run-time are needed [1b]. Thirdly, one categorizes each capability from Step 1b depending on whether it can be realized on existing IT devices or not. On this basis, one compiles a list of components of the legacy system to be supervised by run-time monitors [1c] and a list of devices (i.e., sensors and actuators) to be added to the legacy system [1d]. Finally, each use case is modified according to the requirements [1e].

Decomposition [Step 2]: Firstly, one decides at which level of granularity each monitor shall observe steps of its respective target at run-time. Moreover, one determines which information each monitor shall obtain when its target performs a step and which capabilities each monitor shall have to alter its target's behavior

[2a]. Secondly, one decomposes the modifications of the use cases from Step 1e into suitable modifications of the behavior of individual components of the legacy system. This results in specifications of the local logic of the individual run-time monitors [2b] and of the cooperation requirements to be resolved by communication between monitors [2c]. Thirdly, one decides how the run-time-monitoring framework shall obtain information from each sensor [2d]. One might choose that information shall be pushed by a sensor or needs to be pulled from it. One might choose a point-to-point communication between a sensor and a monitor, making a dedicated monitor responsible for propagating the sensor's information to other monitors. Alternatively, one might choose a multi-cast or broadcast communication, making the sensor's information directly available to multiple monitors. Finally, one decides how the run-time-monitoring framework shall trigger the individual actuators [2e]. For instance, one might determine a dedicated monitor to be responsible for triggering an actuator. If multiple monitors may trigger a given actuator, one needs to apply suitable strategies to avoid possible conflicts by construction or to resolve them at run-time.

Monitor Integration [Step 3]: Firstly, one chooses for each target identified in Step 1c a suitable system layer into which the supervising monitor shall be integrated. Moreover, one decides how the integration into the chosen layer shall be realized [3a]. A monitor's code might be integrated, for instance, by interleaving it with the target's code (in-lining), by running it in a separate process (out-lining), or by a combination of the two (cross-lining). Secondly, one decides on the protocols for communicating between monitors, where such communication is needed [3b]. Thirdly, one integrates the monitoring framework into the legacy system [3c]. Finally, the resulting system is validated [3d].

Device Integration [Step 4]: Firstly, one chooses concrete versions of the devices (sensors and actuators) identified in Step 1d [4a]. Secondly, one decides where and how each of these devices shall be placed physically [4b]. Thirdly, the installation of devices and their integration into the legacy system is performed as specified [4c]. Finally, the resulting system is validated [4d].

Policy Definition and Instantiation [Step 5]: Firstly, one implements the logic of each monitor by defining a local policy [5a] that realizes the desired modifications of the corresponding target's behavior, as identified in Step 2b. Secondly, one implements the cooperation between monitors and of monitors with un-monitored devices, as identified in Step 2c [5b]. Thirdly, one instantiates each monitor with the respective local policy [5c]. Finally, the resulting system is validated [5d].

Intermediate Validation in Steps 3 and 4 A key feature of our approach is that the intermediate steps can be validated by instantiating the run-time monitoring framework using suitable auxiliary policies in Steps 3d and 4d.

To this end, one defines, in Step 3d, for each monitor an auxiliary policy that causes the monitored target to behave equivalently to the target without monitor. The partially modified system can then be tested (or be analytically verified) against the unmodified legacy system for behavioral equivalence. Such a validation can be performed by regression testing, i.e., by using test vectors that had been used for validating the unmodified legacy system.

Similarly, one can define auxiliary policies in Step 4d to test that the newly integrated sensors and actuators work properly. Moreover, one can define policies to test the interaction of the devices with the run-time-monitoring framework. Further auxiliary policies can be defined for testing that monitors obtain the desired information about their targets, that each monitor can alter its target's behavior as desired, and that communication between monitors works properly.

Using auxiliary policies in the validation of these intermediate steps has an additional advantage. Namely, the policies defined in these steps provide a starting point (the no-op policies from Step 3d) and inspiration (the testing policies from Steps 3d and 4d) for the definition of policies in Steps 5a and 5b.

2.2 The Application Scenario

We consider two elevators and three levels of a hypothetical but realistic hospital building: a helipad on the roof, the ground floor, and the basement. We abstract from all other floors of the hospital and also from other possibilities to reach floors from each other, because they are not relevant for our case studies. Access to each of these three levels is restricted to authorized personnel. In particular, patients and visitors are not allowed to enter these areas by themselves.

Hospital Architecture Figure 2 visualizes the floor plans, highlighting entities that are relevant for our case studies and abstracting from details not relevant.

On the top level (Level 15), a helipad is located for the delivery of emergency cases to the hospital. This level can only be reached by the first elevator (L1). On the ground floor (Level 0), there is a dedicated entry for emergency cases that are delivered by car ambulances (at top-left corner in the floor plan). This level can be reached by both elevators (i.e., by L1 and L2). In the basement (Level -1), operating rooms are located in which surgeries of emergency cases are performed. For simplicity, we assume only two operating rooms (O1 and O2), each with a dedicated preparation room (P1 and P2). Usually, emergency cases are transported to a preparation room and are moved to the corresponding operating room via the connecting door. However, if a patient already has been prepared during transport in the ambulance, then this patient may be brought into an operating room directly. To simplify the re-stocking of the operating rooms and the preparation rooms, a supply room (SP) is located in their proximity.

In normal mode, both elevators can be called to all levels by pressing buttons on the outside, and they can be moved to all levels by pressing buttons on the inside. Access to some levels may be restricted to authorized personnel (including Levels -1, 0, and 15). Authorized personnel may put an elevator into emergency mode and may request it in emergency mode. An elevator in emergency mode is under full control of its current operator until this mode is deactivated. In emergency mode, requests for the elevator are ignored to ensure that the transport of patients in critical conditions is not interrupted by less critical cases.

Definition of Selected Use Cases For many routine tasks in a hospital, it is regulated how they may be performed. Such organizational regulations can be defined by workflows or business processes in an established workflow language (like, e.g., BPMN [14]), complemented by textual explanations. To shorten our presentation, we refrain from using such a formal notation in this article.

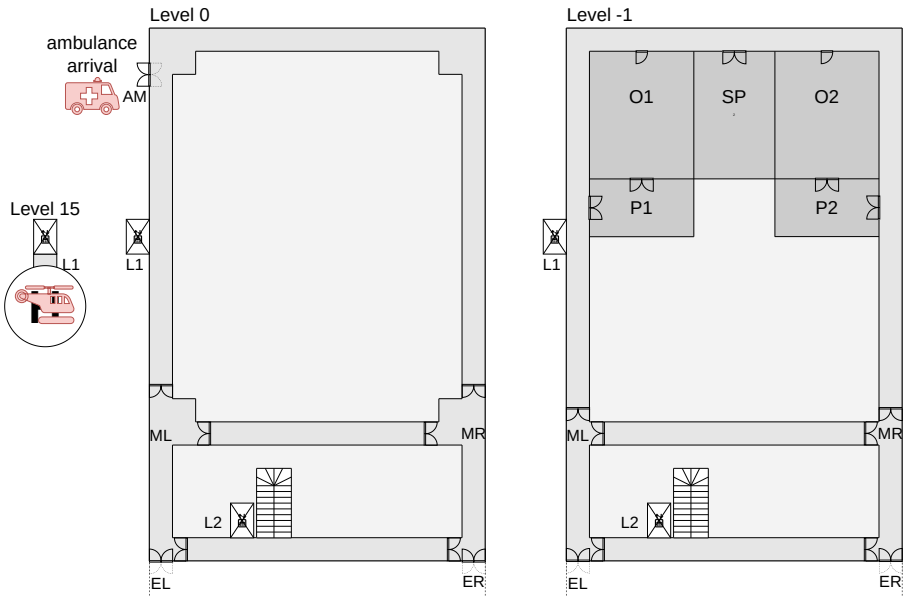

Fig. 2. Hospital building: simplified floor plans for three levels

HELI-1 *and* HELI-2 These use cases are triggered by a heli-ambulance arriving at the helipad with an emergency case, where the patient already has been prepared for surgery during the flight. Which of the use cases is triggered depends on which operating room has been assigned to this emergency by the hospital. The goal is to bring the patient as quickly as possible to this operating room.

The default route is: (1) put the stretcher with the patient on a cart with wheels, (2) use Elevator L1 in emergency mode to move the patient to the basement, (3) after the basement has been reached, transport the patient along the following route L1→O1 for HELI-1 and L1→O2 for HELI-2.[1]

CAR-1 *and* CAR-2 These use cases are triggered by a car ambulance arriving at Entrance AM on Level 0 with a patient not prepared for surgery. Again, the particular use case depends the assigned operating room. The goal is to bring the patient as quickly as possible to the respectively adjacent preparation room.

The default route is: (1) put stretcher with patient on a cart with wheels, (2) transport the patient to Elevator L1 via AM→L1, (3) use L1 in emergency mode to move the patient to the basement, (4) after the basement has been reached, transport the patient along L1→P1 for CAR-1 and L1→P2 for CAR-2.

RE-SUPPLY This use case is triggered at regular intervals and in case the supply in the supply room SP has reached a critical level. The goal is to ensure sufficient supply in SP for quickly re-stocking the two operating rooms between surgeries.

In this use case, the supply cart enters the floor plan from the outside at Level −1 using entry EL carrying the needed material. The default route is: (1) push the supply cart along the route EL→ML, (2) continue along ML→L1→SP.

[1] We use L1→O1 to denote the shortest path from L1 to O1 according to the floor plan. To denote the next shortest route, we disambiguate by using L1→ML→MR→O1.

RE-LOCATE-1 *and* RE-LOCATE-2 These use cases are triggered when a stretcher shall be removed from an operating room and be moved to some other part of the hospital. Which use case is chosen depends on the location of the stretcher. The goal is to re-locate the empty stretcher to its destination.

The default route for RE-LOCATE-1 is: (1) push the stretcher out of O1, (2) move it along O1→L1→L2. (3) use Elevator L2 to transport the stretcher to the destination level in the hospital. The default route for RE-LOCATE-2 is: (1) push the stretcher out of O2, (2) move it along O2→MR→L2. (3) use Elevator L2 to transport the stretcher to the destination level in the hospital.

Note that the above use cases comprise ones that are more critical than others. Logistics within a hospital typically distinguish between high-priority tasks (or emergencies) and low-priority tasks (or normal operation). Our use cases HELI-1, HELI-2, CAR-1, and CAR-2 correspond to high-priority tasks. We view the use cases RE-SUPPLY, RE-LOCATE-1, and RE-LOCATE-2 as low-priority tasks.[2]

3 Re-Engineering Case Studies

We perform three case studies in the hospital scenario from Section 2.2. In each case study,we perform a systematic re-engineering using our method. Each re-engineering is motivated by an incident resulting from resource conflicts (e.g., by the use of an elevator or of a section of an aisle) encountered in daily operation. We consider conflicts between tasks of different priorities in the first two case studies and a conflict between two high priority tasks in the third case study.

Throughout this section, we remain at a conceptual level. We skip steps involving implementation (i.e., Steps 3b, 3c, 4a, 4c, and 5c) and validation (i.e., Steps 3d, 4d, and 5d). The starting point of each case study is our hypothetical hospital as the legacy system and a requirement motivated by some incident.

3.1 Preventing Avoidable Delays of High-Priority Tasks

The transportation of a patient from the helipad to an operating room and a re-supply of the supply room might interfere with each other. Consider an incident where a patient in critical condition arrived via the heli-ambulance and shall be transported to the second operating room (i.e., use case HELI-2). Shortly before, a re-supply was ordered (RE-SUPPLY). Unfortunately, the supply cart broke down on Level −1 between P1 and O1, blocking the default route of HELI-2. This caused a delay of the transportation of the critical patient to O2.

To avoid such incidents in the future, hospital management decided to prevent the use of the shared resource (i.e., the section of the aisle between L1 and O2) in use case RE-SUPPLY if this resource is about to be needed in one of the high-priority use cases HELI-1 or HELI-2. To signal to the supply cart that it must not proceed, a traffic light shall be installed, signaling red if the supply cart must not enter the critical section of the aisle, and signaling green, otherwise.

We illustrate how to apply our MBRE Method in this re-engineering.

[2] In case of an urgent supply shortage, delivering this supply deserves higher priority. We refrain from specifying a use case for such a high-priority task in this article.

Global Analysis [Step 1]: The use cases HELI-1, HELI-2, and RE-SUPPLY are relevant [1a]. For enforcing the requirement, one needs sufficient information for deciding whether the aisle between L1 and O2 must be reserved for HELI-1 or HELI-2, and whether this critical section may be made available to the supply cart. Finally, a possibility is needed to alter the behavior of the supply cart such that it does not enter the critical section when reserved [1b]. We choose to use information retrievable from the state of Elevator L1 as the basis for deciding whether the critical section must be reserved or not. This information about L1's state is available to L1's controller and, hence, can be accessed by a run-time monitor added to this controller [1c]. Thus, there is no need to add sensors to the legacy system.[3] The requirement specification already mandates how to block the critical section, namely, by a combination of an actuator (traffic light) and an organizational policy (supply cart must not proceed if the traffic light signals red). We decide to complement the traffic light with a simple controller that acts as communication point for run-time monitors [1d]. We introduce adaptations of the use cases HELI-1, HELI-2, and RE-SUPPLY in the following [1e]:

HELI-1′ *and* HELI-2′ are triggered in the same way as HELI-1 and HELI-2, respectively. The goals of the original use cases also remain unmodified.

Step 2 of the default routes is refined, resulting in: (1) put the stretcher with the patient on a cart with wheels, (2a) enter L1, (2b) use L1 in emergency mode to move the patient to the basement, (2c) exit L1, and (3) transport the patient along the route L1→O1 for HELI-1 and L1→O2 for HELI-2.

RE-SUPPLY′ has the same trigger and goal as RE-SUPPLY. One action and two preconditions are added to the default route, resulting in: (1) push the supply cart along EL→ML, (2a) wait if the traffic light signals red, (2b) if the traffic light signals green, continue along ML→L1→SP.

Note that the supply cart waits at ML if the traffic light signals red. This decision is motivated by an existing organizational policy, namely, to not wait unnecessarily on aisles to avoid congestion, but rather to wait at intersections.

Decomposition [Step 2]: We decide that the critical section of the aisle shall be reserved for HELI-1 or HELI-2 when Step 2a of the respective use case occurs,[4] In addition, we decide that the critical section may be unblocked after Step 2c of HELI-1 or HELI-2 has been completed. Thus, the monitor added to the controller of L1 must be able to observe at which level L1 is, in which mode L1 is, which buttons are pressed to move L1, whether L1 is in use, and whether the doors of

[3] Should it not be possible to add a run-time monitor to the elevator's controller, then a different design decision must be made at this step. For instance, a run-time monitor could alternatively be added to a system managing the arrivals and departures on the helipad. Another alternative would be to add sensors elsewhere, e.g., within the elevator or onto the stretchers used for patients arriving at the helipad.

[4] This is not the only sensible design choice. Alternatively, one could reserve the critical section earlier (e.g., when L1 is called to Level 15) or later (e.g., when L1 is in emergency mode and passes Level 7 on its way to the basement). Which choice is best depends on multiple factors such as the speed of the elevator, the speed of the supply cart, and the willingness to take risks, and the acceptability of delays.

L1 are open or closed. This information is also needed by the elevator's controller and, hence, can be retrieved from it. The monitor does not need any capability to alter the elevator's behavior [2a]. The monitor tracks the state of the elevator and decides whether to send signals to the traffic light. The monitor signals to reserve the critical section if L1 is on Level 15, in emergency mode, and requested from the inside to move to the basement. The monitor signals to unblock the critical section if L1 reached the basement, returned to normal mode, and its doors have closed. We decide to complement the traffic light with a simple controller that acts as communication point for the run-time monitor added to the elevator's controller [2b]. We decide to use a point-to-point communication between the monitor and the controller of the traffic light [2e]. Since there is only one monitor and no sensors were added, nothing needs to be done in Steps 2c and 2d.

Monitor Integration [Step 3]: We decide to integrate the monitor by in-lining its code into the controller code [3a]. As stated before, we abstract from implementation issues and validation [3b, 3c, and 3d].

Device Integration [Step 4]: We decide to place the traffic light on the wall on the basement at ML, which is suitable for the default route of Re-Supply′. Again, we abstract from implementation and validation [4a, 4c, and 4d].

Policy Definition and Instantiation [Step 5]: We define the policy of the monitor on the elevator's controller by a finite automaton with two states, *restricted* and *unrestricted*, where *unrestricted* is the initial state. If L1 is on Level 15 in emergency mode and requested to move to the basement, then this causes a transition to state *restricted*. If L1 is on the basement, in normal mode, and the doors have been closed, then this causes a transition to state *unrestricted*. A transitions to state *restricted* causes a signal to turn red to the traffic light controller, and a transition to state *unrestricted* causes a signal to turn green [5a and 5b]. Again, we abstract from implementation and validation [5c and 5d].

In this case study, we illustrated how our MBRE Method can be used for a systematic re-engineering. After filling in the implementation and validation steps, one arrives at a distributed IoT system that satisfies the requirement that motivated the re-engineering. Note that the run-time monitor added to the elevator's controller may be re-used in any future re-engineering. For instance, this monitor could be re-used in the case studies presented in Sections 3.2 and 3.3.

3.2 Eliminating Unnecessary Interferences between Tasks

The re-engineering in Section 3.1 was motivated by an accident (supply cart broke down) in one use case (i.e., Re-Supply) that caused a delay in another use case (i.e., Heli-2). Such interferences between use cases might also happen without accidents, and they might cause delays for both use cases. Consider an incident where a patient is transported from the helipad to the second operating room in Heli-2 and a stretcher is removed from the first operating room in Re-Locate-1. Assume that the stretcher with the patient has just left Elevator L1, and the empty stretcher has just left O1. That is, both stretchers are in

the region between L1 and O1 moving towards each other. Although the two stretchers should fit next to each other in this part of the aisle, in principle, they collided because they passed each other without slowing down.

This incident can be addressed in a similar way as the incident in Section 3.1. A key difference is that a sensor needs to be added to realize the blocking of the critical section of the aisle. Because the two stretchers move in opposing directions, the critical section must be blocked longer than in Section 3.1 to avoid a collision. A sensor needs to be added because none of the existing IT devices has sufficient information to decide that the critical section may be unblocked.

Global Analysis [Step 1]: Like in the previous case study, we decide to place a monitor on the controller of L1 [1c] and use a traffic light in combination with an organizational policy to block the critical section [1e]. The traffic light will be placed in O1 [4b]. It signals red if the door of the room must remain closed because the critical section is currently blocked for HELI-2. Accordingly, use case RE-LOCATE-1 is modified by adding that the traffic light signals green as a precondition to Step (1) [1e]. A sensor is needed to learn that the critical section can be unblocked because the patient transport has left it [1d].

Decomposition [Step 2]: We decide to give the monitor at L1 the final say about the blocking and unblocking of the critical section [2b]. This means that the monitor must receive information from the added sensor [2d] and must be able to trigger state changes of the traffic light [2e].

We skip all other steps of the MBRE Method. They can be performed analogously to Section 3.1 due to the deliberate similarities between the case studies.

In this re-engineering case study, we provided an example for when a sensor needs to be added to a legacy system. Moreover, this case study offers the possibility to re-use previously integrated monitors. That is, the monitor at L1 could be re-used from a prior re-engineering, e.g., the one presented in Section 3.1.

3.3 Progressing High-Priority Tasks Despite Collisions

The transportation of patients from the helipad might interfere with the transportation of patients delivered by car ambulance. Consider an incident, where L1 is transporting a patient from the helipad to the basement when a patient arrives at L1 on Level 0 for transportation to an operating room. The delay in the transportation of the latter patient was avoidable, because L2 could have been used for reaching the basement instead of waiting for L1. As a reaction to this incident, hospital management requested a solution for re-routing patients who arrived by a car ambulance from L1 to L2, if L1 is in use and the alternative route likely results in a speed up of the transport. In particular, the availability of Elevator L2 shall be ensured before a patient transport is re-routed to it.

Again, we apply our MBRE Method in this re-engineering case study.

Global Analysis [Step 1]: The use cases CAR-1, CAR-2, HELI-1, and HELI-2 are relevant [1a]. For enforcing the requirement, one needs sufficient information about the state of both elevators to decide whether patients arriving on the

ground floor shall be re-routed to L2. Moreover, an ability is needed for remotely calling L2 to the ground floor and for reserving L2 for the re-routed transport. Finally, an ability to trigger the use of the alternative route is needed [1b]. We retrieve the needed information from the controllers of L1 and L2 by two monitors running on these controllers [1c]. We decide to add a display for signaling that the alternative route shall be taken. Again, no sensors need to be added [1d]. With our design choices, HELI-1 and HELI-2 can remain unchanged. We introduce adaptations of the use cases CAR-1 and CAR-2 in the following [1e]:

CAR-1′ *and* CAR-2′ have the same trigger and goal as CAR-1 and CAR-2, respectively. We modify the default routes by refining Step 3, and we add an alternative route to each use case. The modified default routes are: (1) put stretcher with patient on a cart with wheels, (2) transport the patient to Elevator L1 via AM→L1, (3a) request L1 in emergency mode, (3b) wait until L1 can be used or until the display signals to use the alternative route, (3c) if L1 is available, use it to move the patient to the basement, (4) after the basement has been reached, transport the patient along L1→P1 for CAR-1 and L1→P2 for CAR-2.

The following alternative routes may be used after Step 3b: (3c) if L1 is not available and the display signals to re-route to L2, then transport the patient to Elevator L2 via L1→L2 on Level 0, (3d) use L2 in emergency mode to move the patient to the basement, (4) after the basement has been reached, transport the patient along: L2→L1→P1 for CAR-1 and L2→MR→P2 for CAR-2.

Decomposition [Step 2]: When L1 is requested from the ground floor in emergency mode, we consider a re-routing if the following condition is fulfilled:

COND-1: L1 is in use, in emergency mode, and located at or above Level 4.[5]

To trigger a re-routing the following condition must be fulfilled in addition:

COND-2: L2 is available on the ground floor and reserved.

We use MON-1 and MON-2 to refer to the monitors running on the controller of L1 and of L2, respectively. MON-1 must be able to determine whether L1 is in use, at which level L1 is, and L1's mode. MON-1 must also be able to notice when L1 is called from the basement in emergency mode. MON-1 does not need any abilities to alter the behavior of L1. MON-2 must be able to determine at which level L2 is and whether it is in use. In addition, MON-2 must be able to call L2 to the ground floor and to reserve it for a patient transport [2a].

MON-1 and MON-2 track the state of L1 and L2, respectively. We decide that MON-1 shall have the final say about whether a patient transport is re-routed. Thus, MON-1 needs to retrieve the information about the state of L2 needed for this decision from MON-2. If L1 is requested from the ground floor in emergency mode and COND-1 is fulfilled, then MON-1 asks MON-2 to establish COND-2. To fulfill COND-2, MON-2 calls L2 to the ground floor (unless the elevator is already available on the ground floor), reserves L2, and signals to MON-1 that L2 is reserved. In addition, MON-2 blocks all requests for L2 until the patient has been transported to the basement [2b and 2c]. If

[5] There are manifold alternatives to defining this pre-condition. We perform the re-engineering such that COND-1 can be easily adapted to a different pre-condition.

COND-1 and COND-2 are fulfilled, then MON-1 triggers the display to show a message that the transport shall be re-routed [2e]. Since no sensors were added, nothing needs to be done in Step 2d.

Monitor Integration [Step 3]: We decide to integrate the monitors by in-lining their code into the code of the controllers for L1 and L2 [3a]. We abstract from implementation issues and validation [3b, 3c, and 3d].

Device Integration [Step 4]: We place the display for re-routing next to L1 on the ground floor, as this fits the location of the patient in CAR-1′ and CAR-2′. Again, we abstract from implementation and validation [4a, 4c, and 4d].

Policy Definition and Instantiation [Step 5]: We define a simplified policy for MON-1 by a finite automaton with three states *default*, *enable*, and *trigger*, where *default* is the initial state. If L1 is requested from the ground floor in emergency mode and COND-1 is satisfied, then a transition to *enable* occurs. If COND-2 and COND-1 are satisfied, then a transition to *trigger* occurs. Should COND-1 be violated before, then a transition to *default* occurs. A transition to *enable* causes a request to MON-2 to establish COND-2. A transition to *trigger* causes a signal to the display to show a message that the transport shall be re-routed. After this message has appeared, a transition to *default* occurs.

This policy clarifies how the cooperation with MON-2 and the triggering of the display can be specified. To better clarify these aspects, we omitted how the state of the display and a reservation of L2 are reset. We assume it to be clear at this point how to enrich the the policy with such additional aspects and also how to define a policy capturing the logic of MON-2 [5a and 5b].

Again, we abstract from implementation issues and validation [5c and 5d].

In this re-engineering case study, we have illustrated how to use our MBRE Method when multiple run-time monitors cooperate with each other. Note that already existing monitors could be re-used. Monitors can be re-used by replacing their current policies or by integrating the new policy with the current policy.

4 Implementation and First Performance Evaluations

Having a legacy system is a prerequisite for any re-engineering. We implemented parts of the hospital scenario from Section 2.2 in our testbed for IoT applications. This test-bed features a spectrum of hardware platforms, ranging from simple controllers to more powerful controllers to laptops and servers.

Having an implementation of the hospital scenario under our control enables us to perform repeatable experiments. In our implementation, we chose an architecture that resembles the architecture of our hypothetical hospital, i.e., different components of the legacy system are realized by different hardware components in our test-bed. This approach shall facilitate meaningful experimental results and shall ease a transfer of insights from the test-bed to real-world systems.

In Section 4.1, we describe the implementation of selected components of the legacy system in our test-bed. We also explain how the sensors and actuators

from our three case studies in Sections 3.1, 3.2, and 3.3 can be realized, which corresponds to Step 4a of the MBRE Method that we skipped in Section 3.1.

In Section 4.2, we show how to integrate run-time monitors and how to instantiate them with policies at the example of the monitor on the controller of L1 from Section 3.1. This corresponds to Steps 3c and 5c of our MBRE Method. In Section 4.3 we investigate the overhead caused by our implementations of run-time monitors. Interestingly, we found that this overhead differs for different hardware platforms not only in absolute numbers, but also when normalized.

4.1 A Test-bed for the Hospital Scenario

We used two controller boards in our implementation of the legacy system:

PICO platform consisting of a Raspberry Pi Pico W with an RP2040 ARM Cortex-M0+ micro-controller with a clock speed of up-to 133 MHz, 264 kB on-chip RAM, and a Wi-Fi module

4B platform consisting of a Raspberry Pi 4B single-board computer with a quad-core 1.8 GHz BCM2711 ARM System on a Chip (SoC) and 8 GB RAM

The 4B platform is more powerful than the PICO platform, in both, processing speed and available RAM. In particular, the 4B platform is powerful enough to run a Java virtual machine, while the PICO platform is not. However, the PICO platform is powerful enough to run a Python interpreter. To provide manual input, we added a touch-screen to the 4B platform. Since the PICO platform is not powerful enough to drive this touch-screen, we added physical buttons. Both platforms enjoy wide-spread use and are popular for IoT applications.

We implemented the controller for elevators on both platforms to enable experimental comparisons. We first describe the I/O interfaces:

PICO We used physical push buttons to model the buttons on the outside and inside of the elevator. This includes buttons for requesting the elevator from the outside, for directing the elevator to a particular level from the inside, for switching the mode of the elevator from the inside, and for requesting the elevator in emergency mode from the outside.
We used an LED to indicate the mode of the elevator (normal or emergency). Further information about the internal state of the elevator can be retrieved from the serial console like, e.g., the mode, the current level and the destination level to which the elevator is moving.

4B We attached a 7-inch touch-screen and implemented buttons virtually. The range of buttons supported is the same in both implementations. We used the touch screen also to display the internal state of the monitor.

The logic of the elevator controller is implemented in Python on PICO. It consists of one module, 22 functions, and it has 160 lines of code (LoCs). We used Java for the implementation on 4B. This implementation consists of one class, 24 methods (11 private, 10 public, 3 protected), and it has 242 LoCs.

In addition, we used three types of devices as sensors and actuators: LEDs of multiple colors, a motion sensor, and two 7-inch displays. As explained in Section 3, we complemented these devices by controllers (either 4B or PICO)

that act as communication points. We chose Python to program the logic on these controllers and HTTP for the communication with run-time monitors.

Figure 3 summarizes the mapping of devices mentioned in Section 3 to hardware devices in our test-bed. We chose the PICO platform as controller for all devices, except for the display, for which we chose the more powerful 4B platform.

Sect.	Device	Sensor	Actuator	Hardware
3.1	Traffic Light		✗	red and green LED
	Controller			PICO platform
3.2	Traffic Light		✗	red and green LED
	Controller			PICO platform
3.2	Sensor	✗		motion sensor
	Controller			PICO platform
3.3	Display		✗	7-inch display
	Controller			4B platform

Fig. 3. Mapping of devices from the re-engineering case studies to our test-bed.

4.2 Two Monitor Implementations

We implemented the run-time monitor on the PICO platform as a Python module. The monitor is activated when its target executes a Python function. As an optimization, the monitor is only activated by function calls that are relevant for the elevator's state and behavior. For each function call, the monitor obtains information about the actual arguments of the function call. The monitor is also able to retrieve further information from the state of the target. The monitor can retrieve additional information by communicating with sensors and other run-time monitors. The monitor can influence the behavior of its target by modifying its state, and it can trigger actuators. The monitor is parametric in the policy. To support efficient decisions based on a given policy, the relevant information about a function call is packaged into a dedicated data structure.

Our implementation of the run-time monitor for the 4B platform has the same structure. However, the monitor is implemented as a Java class, and the run-time monitor is activated when its target executes a Java method. For a function call, the monitor obtains the arguments and can access the state of the target to retrieve further information. To support efficient decision making, the relevant information is packaged into an object from a dedicated Java class.

The PICO platform only offers one layer for integrating a run-time monitor, because there is no operating system or virtual machine. We chose the in-lining technique for coupling a monitor with the source code of its target, i.e., the code implementing an elevator controller. In-lining the monitor into machine code would not be a sensible option as Python is interpreted and not compiled.

For the 4B platform, we also used in-lining on the source-code level to integrate the run-time monitor into the implementation. In principle, this platform offers further layers into which a monitor could be integrated, e.g., the operating system and the Java virtual machine. Since our goal in this section is a fair

performance comparison, we used the same abstraction level (i.e., source code) and technique (in-lining) for integrating the monitor into the code of the target.

We specified the policy as a finite state machine with two states as described in Step 5a of the case study in Section 3.1. The only notable difference between the two platforms is the programming language in which the policy is defined, namely in Python for PICO and in Java for 4B. In addition to this policy, we specified no-op policies for both platforms, i.e., in Python and in Java.

Due to our design choices, the instantiation of a generic run-time monitor with a given policy technically corresponds to inserting the code specifying the policy into the monitor's code. This principle applies to both platforms.

The behavior of a monitored system depends on the target component of the legacy system (e.g., the elevator controller), the selection of functions or methods that activate the monitor, and the policy. The design choices we made in this section aimed for a quick comparison of the performance of monitor implementations on different platforms. We will re-visit the design choices in the future, in particular, to ensure that policies can, indeed, be exchanged at run time, which is not yet the case with our prototypical implementation.

4.3 Performance Evaluation and Comparison

We focus on the Steps 2a and 2c of HELI-1′ and HELI-2′ in our performance evaluation. Our experimental setup is described at the end of this section.

In Step 2a, a patient transport entered the elevator on Level 15, the elevator is in emergency mode, and is requested internally to move to the basement. The pressing of the button inside the elevator causes a signal to the elevator's controller, which results in a call of the function `internalCall` on the PICO platform and in a call of the method `internalCall` on the 4B platform. In the body of the function/method, the state of the controller is updated to reflect the change of the elevator's state. We measure the time from when the function/method `internalCall` is called to when its execution is completed.

In Step 2c, the patient transport left the elevator on the basement and the doors are closing. This results in a call of the function `doorClose` by the elevator's controller on the PICO platform and of the method `doorClose` on the 4B platform. In the body of the function/method, the state of controller is updated to reflect the change of the elevator's state. We measure the time from when `doorClose` is called to when its execution is completed.

Steps 2a and 2c appear quite similar, and one might expect similar performance results. However, the controller's internal data structures that need to be updated in these steps have different performance characteristics. Since the internal update is faster for Step 2c, we present our evaluation of it first.

For PICO, our performance evaluation results for Step 2c and Step 2a are summarized in the table on the left-hand side and on the right-hand side, respectively, of Figure 4. Note how substantial the difference of the run time for the original implementations is in the tables (see first row of the two tables). Step 2c takes 32.97μs, while Step 2a takes 68.76μs. This is the result of the difference in the data structures that are updated. The second row of both tables shows

the run time if the elevator's controller is supervised by a run-time monitor, the delay caused by the monitor (absolute overhead), and the delay normalized by the run time of the un-monitored controller (relative overhead). Note that the absolute overhead is slightly larger for Step 2a. This is due to the differences in the parts of the policy that need to be evaluated in these two steps.

For 4B, our performance evaluation results are summarized in Figure 5. Again, the difference of the run time for the original implementations is substantial. Step 2c takes $0.74\mu s$, while Step 2a takes $2.61\mu s$. This is the result of the difference in the data structures that are updated. The second row of both tables shows the run time if the elevator's controller is supervised by a run-time monitor, the delay caused by the monitor, and the delay normalized by the run time of the un-monitored controller. Note that the absolute overhead is larger for Step 2a. Again, this is due to the differences in the relevant parts of the policy.

(2c)	run time	absolute overhead	relative overhead
original	32.97μs		
revised	231.17μs	198.21μs	601.26%

(2a)	run time	absolute overhead	relative overhead
original	68.76μs		
revised	288.00μs	219.25μs	318.87%

Fig. 4. Time measurements on PICO for (2c) `doorClose` and (2a) `internalCall`

(2c)	run time	absolut overhead	relative overhead
original	0.74μs		
revised	2.95μs	2.21μs	296.40%

(2a)	run time	absolut overhead	relative overhead
original	2.61μs		
revised	5.82μs	3.22μs	123.39%

Fig. 5. Time measurements on 4B for (2c) `doorClose` and (2a) `internalCall`

A comparison of the run time across the two platforms reveals a substantial difference for the original implementations ($32.97\mu s$ vs $0.74\mu s$ for 2c and $68.76\mu s$ vs $2.61\mu s$ for 2a). This difference shows how much more powerful the 4B platform is, in particular, as Linux and a Java virtual machine are running on this platform, while the code runs on the bare PICO platform. Accordingly, the absolute overhead on the PICO platform exceeds the absolute overhead on the 4B platform roughly in the same order of magnitude.

What we found rather surprising is the substantial difference in the relative overhead across the two platforms (601.26% vs 296.40% for 2c and 318.87% vs 123.39% for 2a). We have not found a convincing explanation yet for where this difference originates from. As these are normalized numbers, the difference in the power of the platforms does not serve as a natural explanation.

Experimental setup and measurements. For our performance measurement, we disabled the communication of the monitor with sensors and actuators to avoid that the measurements are corrupted by non-deterministic delays due to network communication. Technically, we replaced the function/method calls for such communication with stub functions with an empty body. Consequently,

the measured performance overhead originates only from the invocation of the monitor, from the policy evaluation, and from the monitor's internal logic.

For our experiments with the Java implementation on 4B, we employed the Java Microbenchmark Harness (JMH) [2], which is popular and part of the OpenJDK project. Compared to JMH's default parameters, we increased the number of warm-up and measurement iterations to 10 (from 5) with a time of $10s$ (from $5s$) each. This resulted in more consistent measurements. Moreover, we disabled some JVM optimizations (JIT compilation and some intrinsics).

Due to the absence of a similar framework for the PICO platform, we implemented our own measurement functionality. In this implementation, we used a strategy analogous to the one of JMH. In particular, we also warm up the Python interpreter until it has stabilized before taking measurements.

5 Related Work

Integrating run-time monitors to create flexibility in system design shares its motivation with other approaches to making software-based systems more flexible. For instance, software product lines [16] address variability by distinguishing components appearing in all variants, in some variants, or only in individual components. By managing them throughout the software life cycle, re-usability of variation can be maximized, and the use of software product lines has proven rather beneficial in software development [13]. Dynamic software product lines offer the possibility to bound variation points at run-time [7], which bears similarities to adaptations caused by run-time monitors. While the construction of software product lines by re-engineering legacy systems has received attention by the research community (e.g., [11, 4]), the integration of run-time monitoring into legacy systems, surprisingly, has received relatively little attention so far. To our knowledge the MBRE Method proposed in this article is the first run-time-monitoring-based re-engineering method for distributed systems.

MAPE-K offers an approach for engineering systems that are adaptable and possibly self healing [9]. To achieve flexibility, a control loop with four steps is used: monitor, analyse, plan, and execute. Different instances of the control loop share a common knowledge base, i.e., have a persistent state. Our MBRE Method might provide an alternative to MAPE-K with lower overhead. However, we have not yet performed experiments on how the control loop of MAPE-K compares to our selective integration of run-time monitors.

Use cases play a crucial role in our MBRE Method. The adaptation of selected use cases in Step 1e can be viewed as a special case of business-process improvement [3]. While monitoring plays a central role in the execution of business processes by engines like, e.g., Camunda BPM [1], such monitoring focuses mostly on obtaining metric data about the process execution. The monitors used in our method not only observe but can also influence their target's behavior.

In IT-security research, the ability of run-time monitors to influence a target's behavior evolved incrementally, starting with security automata [19], which prevent policy violations by terminating a target's run. Edit automata [12] have

the additional ability to suppress steps, to insert steps into a run, and to modify steps. Service automata [6] add the ability to coordinate the actions of decentralized monitors in distributed systems. The enforcement capabilities of dynamic enforcement mechanisms is studied more generally in [8] and [5].

Usage control [15] and distributed usage control [17] can be viewed as generalizations of access control in centralized and distributed systems, respectively. Usage control bears close ties to run-time monitoring. For instance, in [18], a run-time monitoring framework is proposed for usage control in business processes. In recent years, the integration of business process languages and business process engines with the IoT and cyber-physical systems, more generally, has received much attention [20, 10]. In such a combination, the business process engine can act as a run-time monitor of the entire system, but this is centralized monitoring. We are not aware of any solutions that integrate run-time monitors in a decentralized fashion with the ability to cooperate, like in our approach.

6 Conclusion

To our knowledge, the MBRE Method is the first systematic method for run-time-monitoring-based re-engineering of distributed systems. We illustrated our method in three case studies in a hospital scenario in Section 3, where the goal was to avoid problematic incidents experienced in daily operation. For presentation purposes, we chose re-engineering tasks of low conceptual complexity. The need for a systematic re-engineering becomes clearer when considering more complex re-engineering tasks. Our method does not mandate the granularity of a re-engineering, it can be applied for major changes and also for small changes.

When applying the MBRE Method repeatedly, monitors that have been integrated can be re-used in subsequent applications of the method. Imagine that our example legacy system were re-engineered three times by performing the case studies from Section 3 subsequently. In this case, the monitor added to the controller of L1 in the first re-engineering case study could be re-used in the subsequent two case studies. However, it might also happen that a monitor needs to be replaced, e.g., because the monitor observes steps of its target at a too coarse granularity to define a policy that establishes the given requirements.

Based on our implementation of selected components, we presented first results regarding the performance overhead caused by run-time monitors and made an interesting observation that deserves further investigations. We plan to experimentally study the trade-off between improvements gained by monitoring-based re-engineering and the overhead caused by the monitors also in case studies of realistic size. Our test-bed for IoT applications from Section 4 will provide sufficient flexibility for performing case studies also in other application domains.

Acknowledgements. We thank the anonymous reviewers for their constructive feedback and suggestions. This research work was supported by the National Research Center for Applied Cybersecurity ATHENE. ATHENE is funded jointly by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research and the Arts.

References

1. Camunda. https://camunda.com/, [Online; accessed May-31-2024]
2. Java Microbenchmark Harness (JMH). https://openjdk.org/projects/code-tools/jmh/, [Online; accessed Jul-17-2024]
3. Adesola, S., Baines, T.: Developing and evaluating a methodology for business process improvement. Business Process Management Journal **11**(1), 37–46 (2005)
4. Assunção, W.K.G., Lopez-Herrejon, R.E., Linsbauer, L., Vergilio, S.R., Egyed, A.: Reengineering legacy applications into software product lines: a systematic mapping. Empirical Software Engingeering **22**(6), 2972–3016 (2017)
5. Basin, D.A., Jugé, V., Klaedtke, F., Zalinescu, E.: Enforceable security policies revisited. ACM Transactions on Information and System Security **16**(1), 3 (2013)
6. Gay, R., Mantel, H., Sprick, B.: Service Automata. In: Post-Proceedings of the 8th International Workshop on Formal Aspects of Security and Trust (FAST 2011). pp. 148–163. LNCS 7140, Springer (2012)
7. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic Software Product Lines. Computer **41**(4), 93–95 (2008)
8. Hamlen, K.W., Morrisett, J.G., Schneider, F.B.: Computability Classes for Enforcement Mechanisms. ACM Transactions on Programming Languages and Systems **28**(1), 175–205 (2006)
9. IBM: An architectural blueprint for autonomic computing. Tech. rep. (2006)
10. Kirikkayis, Y., Winter, M., Reichert, M.: A User Study on Modeling IoT-Aware Processes with BPMN 2.0. Information **15**(229) (2024)
11. Laguna, M.A., Crespo, Y.: A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. Science of Computer Programming **78**(8), 1010–1034 (2013)
12. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. International Journal of Information Security **4**(1-2), 2–16 (2005)
13. van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering. Springer (2007)
14. OMG: Business Process Model and Notation (BPMN), Version 2.0.2 (January 2014), https://www.omg.org/spec/BPMN/2.0.2
15. Park, J., Sandhu, R.S.: Towards Usage Control Models: Beyond Traditional Access Control. In: 7th ACM Symposium on Access Control Models and Technologies. pp. 57–64. ACM (2002)
16. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering – Foundations, Principles, and Techniques. Springer (2005)
17. Pretschner, A., Hilty, M., Basin, D.A.: Distributed Usage Control. Commununications of the ACM **49**(9), 39–44 (2006)
18. Pretschner, A., Massacci, F., Hilty, M.: Usage Control in Service-Oriented Architectures. In: Trust, Privacy and Security in Digital Business, 4th International Conference. LNCS, vol. 4657, pp. 83–93 (2007)
19. Schneider, F.B.: Enforceable Security Policies. ACM Transactions on Information and System Security **3**(1), 30–50 (2000)
20. Schönig, S., Ackermann, L., Jablonski, S., Ermer, A.: IoT meets BPM: a bidirectional communication architecture for IoT-aware process execution. Software and Systems Modeling **19**(6), 1443–1459 (2020)