AMon: A domain-specific language and framework for adaptive monitoring of Cyber–Physical Systems

The Journal of Systems & Software 195 (2023) 111507

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

AMon: A domain-specific language and framework for adaptive monitoring of Cyber–Physical Systems☆

Michael Vierhauser [a,*], Rebekka Wohlrab [b], Marco Stadler [a], Jane Cleland-Huang [c]

[a] Johannes Kepler University Linz, LIT Secure and Correct Systems Lab, Linz, 4040, Austria
[b] Chalmers | University of Gothenburg, Department of Computer Science and Engineering, Gothenburg, 41296, Sweden
[c] University of Notre Dame, Department of Computer Science and Eng., Notre Dame, 46617, IN, United States

ARTICLE INFO

Article history:
Received 5 April 2022
Received in revised form 22 July 2022
Accepted 5 September 2022
Available online 22 September 2022

Keywords:
Runtime monitoring
Adaptive monitoring
Domain-specific language
Cyber–Physical Systems

ABSTRACT

Cyber–Physical Systems (CPS) are increasingly used in safety–critical scenarios where ensuring their correct behavior at runtime becomes a crucial task. Therefore, the behavior of the CPS needs to be monitored at runtime so that violations of requirements can be detected. With the inception of edge devices that facilitate runtime analysis at the edge and the increasingly diverse environments that CPS operate in, flexible monitoring approaches are needed that consider the data that needs to be monitored and the analyses performed on that data. In this paper, we propose AMon, a flexible adaptive monitoring framework that supports the specification and validation of monitoring adaptation rules, using a domain-specific language. Based on these rules, AMon automatically generates code for direct deployment onto devices. We evaluated AMon by applying it to TurtleBot Robots and a fleet of Unmanned Aerial Vehicles. Furthermore, we conducted a user study assessing the understandability and ease of use of our language. Results show that creating multiple adaptation rules with our DSL is feasible with minimal effort, and that adaptive monitoring can reduce the amount of runtime data transmitted from the edge device according to the current state of the system and its monitoring needs.

1. Introduction

As Cyber–Physical Systems (CPS), such as autonomous vehicles (Stocco et al., 2020), robotic applications (Mallozzi et al., 2020), and self-adaptive Unmanned Aerial Vehicles (UAVs) (Cleland-Huang et al., 2018; Pereira et al., 2009) are used in largely uncertain and safety–critical environments, ensuring the correct behavior of these systems and their components has become a crucial endeavor. To enable quality assurance, and validate that a running system fulfills its requirements, it is important to adequately monitor its behavior. The collection and analysis of data at runtime, commonly referred to as Runtime Monitoring, has become an increasingly explored research area (Rabiser et al., 2017).

Furthermore, the continuing rise in edge devices introduces the possibility of monitoring at the edge, rather than sending information to a centralized monitoring system. This not only reduces communication overhead, but also decreases the latency between data collection and analysis and is therefore particularly useful for enabling a device to autonomously monitor its behavior in the presence of real-time constraints (Taherizadeh et al., 2018). However, distributed and more flexible monitoring solutions introduce new challenges to monitoring, including specifying properties to be monitored, their frequencies and contexts, and determining whether the monitoring activities will be performed locally (on the device) or centrally. In practice, defining the monitoring behavior is a non-trivial task and many systems use sub-optimal one-size-fits-all monitoring strategies instead of adaptive monitoring.

While some approaches explicitly target adaptive monitoring (Brand and Giese, 2018; Ehlers and Hasselbring, 2011), limited work has explored the adaptation of the runtime monitoring infrastructure itself. In a recent systematic mapping study (Zavala et al., 2019), *adaptive monitoring* has been defined as "the ability a monitoring system has to modify its structure and/or behavior, in order to respond to internal and external stimuli". The study found limited applicability of existing adaptive monitoring frameworks beyond their original application domains, requiring *"more complete, flexible, reusable and generic software engineering solutions for supporting adaptive monitoring"* (Zavala et al., 2019).

To illustrate this, consider the case of UAVs providing support for diverse mission tasks, such as searching for a missing person during a rescue operation (Erdelj et al., 2017). Depending on the state of the mission and environmental factors (e.g., weather or

☆ Editor: Heiko Koziolek.
* Corresponding author.
E-mail address: michael.vierhauser@jku.at (M. Vierhauser).

https://doi.org/10.1016/j.jss.2022.111507

air traffic), different types and amounts of data need to be collected and checked. For example, during take-off, initial startup parameters, such as GPS fix, are of particular interest; whereas during flight, the altitude and battery levels need to be maintained within a certain range to ensure safe operations. This requires the monitoring infrastructure to dynamically adapt according to the system status and to activate or deactivate data collection of specific attributes, and to increase or decrease rates of data collection accordingly.

In this article, we present AMon, an approach for specifying monitoring adaptation rules, and automatically generating adaptive monitors for a target CPS. We propose and evaluate an adaptive monitoring platform that supports the definition of monitoring behavior that can be triggered by state-based or global events. Furthermore, we differentiate between local and central monitoring to reduce the amount of data that is sent to a centralized monitoring system and provide support for the specification of consistent monitoring adaptation behavior through tool support.

We address these requirements through a model-based monitoring framework, contributing: (i) a domain-specific language (DSL) for specifying monitoring adaptation rules; (ii) static validation of the rules to ensure their consistency; (iii) automated code generation for deploying the monitoring framework; and (iv) an evaluation using a CPS for managing UAVs, as well as TurtleBot robots. To evaluate AMon, we conducted experiments and simulations, as well as a qualitative user study involving six participants in order to investigate AMon's understandability and ease of use.

The remainder of this paper is structured as follows. Section 2 presents a motivating example and challenges to adaptive monitoring, while Section 3 provides an overview of our approach. Sections 4 and 5 describe our DSL for adaptive monitoring, validation of the defined rules, generation of monitors, and prototype implementation. In Section 6 we lay out evaluation objectives and then describe the evaluation using a UAV system and ROS-based TurtleBot robots in Section 7. In Section 8 we report results from our user study. Threats to validity are presented in Section 9 and our findings are discussed in Section 10. Finally, related work is presented in Section 11 and the paper is concluded in Section 12.

All data related to the study and the DSL are available in our GitHub repository (AMon, 2022).

2. Motivating example and challenges

We motivate the need for adaptive monitoring through an example of a UAV-based search and rescue mission in an urban area. As multiple UAVs work together and (semi-)autonomously execute the mission, their current state and behavior need to be continuously monitored so that the UAVs and their operators can react to emergent hardware issues, such as battery failures, or safety–critical situations. Given the limited computation and transmission resources of the UAV, and its onboard companion devices, the monitoring load should be selectively increased or decreased according to the current situational needs.

For example, at the start of a mission, UAVs perform initial preflight checks. During this startup phase, the operator observes the UAVs' GPS signal and configuration properties, e.g., by checking whether battery failsafes have been correctly established, or if a geofence has been set to prevent UAVs from flying outside a predefined area. Once preflight checks are completed, these properties become less important and do not need to be monitored.

Additionally, depending on the available computation capabilities of the monitored device, certain monitors can be executed locally, reducing the required bandwidth for data transmission to a central server. Existing monitoring frameworks are commonly coordinated either entirely centrally or entirely locally, but rarely allow for *dynamically changing scope according to the current state of the system*.

For example, once the UAVs take off, their current and target altitudes, and their GPS coordinates become particularly important, especially if multiple UAVs are launched in close proximity to each other. Furthermore, location data may need to be analyzed centrally to prevent early-flight collisions, by detecting conflicting takeoff altitudes or detecting overly close proximity on the launch pad. When the UAVs start flying towards their assigned waypoints, the frequency at which altitude and GPS coordinates are monitored can be reduced, while other checks, such as whether the UAV is following its planned flightpath, can be shifted to the onboard computer.

However, *in addition to this "normal course" of preplanned steps, undesired situations, i.e., "exception courses" can also occur*. For example, during flight, a UAV's onboard sensors might detect another UAV in close proximity, resulting in an increased monitoring frequency for proximity. Furthermore, data should be again monitored centrally, for as long as minimum separation distances are violated. Once the situation is resolved, the monitoring framework can again reduce monitoring frequency and scope to minimize communication overhead and preserve battery.

To support dynamic monitoring, e.g., when UAVs transition between different states while performing a mission, rules need to be defined covering these specific situations or events. This affects both the "regular" states (such as taking off or flying to a waypoint), as well undesired behavior that can occur at any time during the mission. While self-adaptive systems already support similar scenarios, adapting the behavior of the system itself (e.g., through system goal models (Baresi et al., 2010; Muccini et al., 2016)), such adaptations fail to address the need to *change the behavior of the monitors*, adapting how data is collected.

3. The AMon framework

To provide support for dynamic adaptation at the *monitoring level*, we have developed a framework for adaptive monitoring. AMon focuses on small autonomous robotic systems and can be integrated into existing systems, such as self-adaptive systems using the MAPE-K feedback loop.

3.1. Scope of our approach

Fig. 1 provides an overview of how AMon can be embedded into a distributed self-adaptive system using the MAPE-K framework (Kephart and Chess, 2003), which supports monitoring (M), analyzing monitored data (A), planning adaptations (P), and executing (E) them. Self-adaptive systems typically consist of multiple, potentially distributed, subsystems and components (Weyns et al., 2013). AMon supports users in generating an API that enables local monitoring components to send monitored data to an adaptive monitoring component (labeled as Monitoring Comp. in Fig. 1). In terms of adapting monitoring needs, AMon facilitates the dynamic adjustment of the scope, period, and status of the monitoring properties depending on the current context. The scope hereby indicates whether data should be forwarded to the centralized monitoring component of the overall system or analyzed locally. The status indicates whether a property should currently be analyzed or not.

Furthermore, different paradigms for performing runtime monitoring exist, influencing the interaction of the monitoring component(s) with the system to be monitored. Push-based monitoring components play a passive role in receiving data

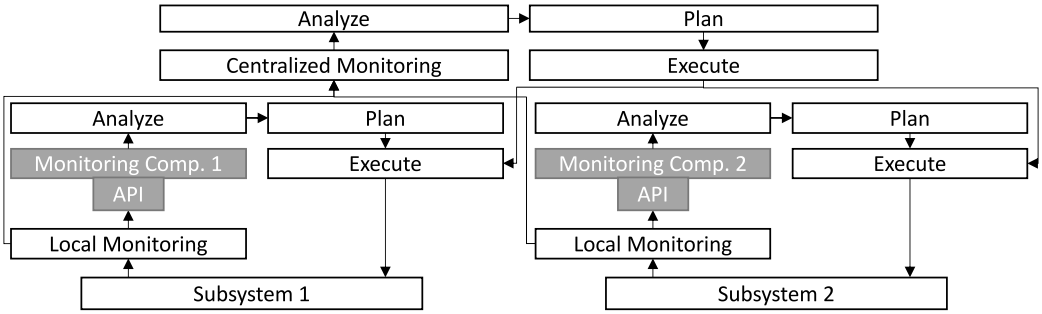

Fig. 1. Overview of how AMon can be embedded into distributed self-adaptive systems.

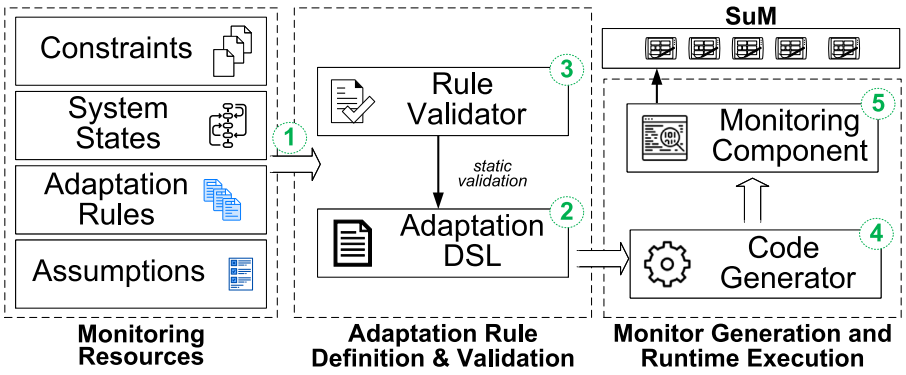

Fig. 2. High-level overview of the main components of our AMon framework.

and do not actively interact with the system; whereas pull-based components actively request data from the system under monitoring (Rabiser et al., 2019). To reduce the user effort of implementing adaptive monitoring, we adopted a passive approach that provides a simple API for pushing data to the adaptive monitoring component. In this approach, the monitor is responsible for adapting the scope and period of each property, and users adopting our framework do not need to implement additional interfaces and components in order to dynamically request data.

However, this decision requires certain assumptions to be made about the system's performance. The monitoring component assumes that the time between properties being pushed does not exceed the specified maximum period, e.g., at least every second, so that the monitoring requirements defined in the rules can be fulfilled.

For example, given a rule that the location data for UAVs is monitored once per second, the system must provide an updated location at least every second. To check that such assumptions hold, we provide validation capabilities for adaptive monitoring configurations described using our DSL (Section 5.3).

3.2. Framework overview

Fig. 2 provides a high-level overview of the main parts of our monitoring framework. We use a number of monitoring resources as an input for the adaptation rules (1). To apply monitoring rules for different system states, we use a state transition diagram that describes the states and triggers for the state transitions. The states are specific to the system and can be modeled, for instance, as a Papyrus state chart (Lanusse et al., 2009).

The purpose of the state transition diagram is to facilitate the specification of rules that serve as preplanned rules during runtime execution, indicating what system states require which properties to be monitored. Furthermore, adaptation rules must also describe the scopes and periods of monitoring properties for relevant system states.

The remaining monitoring resources (Constraints, Adaptation Rules, and Assumptions) are defined in our *Adaptation Rule DSL (2)*. To ensure consistency of the specified rules, a rule validation component (3) performs static validation, for example, providing a warning if a rule specifies a higher monitoring period for a property than defined in the assumptions for that property.

One novel aspect of our dynamic monitoring approach is support for automated generation (4) of monitors and their dynamic adaptation when rules are triggered. We leverage model-driven techniques to automatically generate code based on the rules specified in our DSL and the state transition diagram. The generated monitoring component is capable of switching states according to the current context and transition rules, and executing the respective monitoring rules for each state without the need to manually implement adaptive behavior in either the system or monitoring component. The push-based monitoring component can be directly deployed (5) to the system and/or devices (e.g., the flight computer of the UAV) where it will receive data via the generated API. The rules are executed on the edge device, as part of the monitoring component. Internally, the generated monitoring component schedules monitoring data to be sent to either a central monitoring server or a local monitoring component. If data is pushed more frequently than necessary, the component adjusts the frequency according to the currently active rule. At run time, the monitoring component complements the static validation by checking that data is pushed at a sufficient frequency, as specified in assumptions using the Adaptation Rule DSL. If a deviation is detected, the monitoring component can issue warnings.

4. A domain-specific language for specifying adaptive monitors

To facilitate the specification of adaptation rules for a runtime monitoring environment, we created a domain-specific language that provides capabilities for defining different types of adaptation rules and specifying assumptions about the monitoring data.

Based on the scenarios described in Section 2, and on existing literature in adaptive monitoring (Rabiser et al., 2017; Ehlers and Hasselbring, 2011; Al-Shaer, 1999), we include two types of rules: (1) *Preplanned Rules* that are triggered based on different "regular" states of the system, or of its devices (e.g., a UAV being in a "takeoff" state versus "flying"), and (2) *Ubiquitous Rules* that are triggered by exceptions (e.g., a low battery warning) or other factors. In addition to the specification of rules, we also support the description of assumptions about the system and its environment.

4.1. Rules

Rules for Preplanned Adaptation: As previously explained, preplanned adaptations of the monitoring environment may occur as the CPS changes state, and these adaptations may require changing the monitoring location from the edge device (e.g., onboard the UAV) to a centralized monitoring component. Listing 1 shows a Preplanned Rule for monitoring preflight data. Such rules can either apply *globally*, as shown in this example, or to specific devices (e.g., a group of UAVs performing a joint mission, or even individual devices identified by their respective id). The *Context* element in the *Trigger* condition indicates the UAV's current state. Subsequently, each *Monitor* entry denotes a property item (i.e., a message) that is collected and distributed by the monitoring system.

Monitor specifications define the *Scope* i.e., whether the data is processed locally on the device (scope local), or sent to a central server (scope central) for processing or further analysis. The *Period* further specifies how often data is sent from, for example, the edge device to the central server.

```
Rule 'Preflight_Data' applies globally
   Type PREPLANNED
   Trigger Context['initialized'] ENTRY
   Monitor 'Drone.GPS':
      CHANGE_SCOPE: scope central
      CHANGE_PERIOD: every 0.5 seconds
   Monitor 'Drone.StartupChecks':
      CHANGE_SCOPE: scope central
      CHANGE_PERIOD: every 1 seconds
```

Listing 1: Preplanned Rule – Preflight Checks.

This preplanned rule applies from the time the UAV transitions to the initialized state until its preflight checks are completed and it switches to another state.

Rules for Ubiquitous States: In addition to the preplanned course of events, modeled in a state transition diagram, deviations from the intended behavior may occur. In such cases, the monitoring infrastructure must adapt the monitors to provide sufficient information according to the current state of the system. For example, as specified in Listing 2, a sudden drop in a UAV's battery voltage level might require a corresponding increase in monitoring periods whilst remedial actions are taken to preserve energy and ensure the UAV's safe return or landing.

Ubiquitous rules can be triggered at any time, regardless of the state of the system, and will supersede the original adaptation rules for a certain state. For example, given a sudden drop in battery voltage, the monitoring period might be decreased for less essential properties in order to preserve battery, but increased for essential battery data. When the `battery_state_warning` event is triggered, the corresponding ubiquitous rule (cf. Listing 2) supersedes the monitoring behavior of the UAV's current context. If at some point the ubiquitous state no longer applies (e.g., because the battery warning was a temporary glitch), the UAV resumes its normal mode of operation. In these contexts, an `EXIT` event for a ubiquitous state indicates that the ubiquitous rule should no longer apply, and the monitoring infrastructure

```
Rule 'Battery_State_Warning' applies globally
   Type UBIQUITOUS
   Trigger Event['battery_state_warning'] ENTRY
   Salience 1
   Monitor 'Drone.GPS':
      CHANGE_SCOPE: scope local
      CHANGE_PERIOD: every 3 seconds
   Monitor 'Drone.Battery':
      CHANGE_SCOPE: scope global
      CHANGE_PERIOD: every 1 seconds
```

Listing 2: Ubiquitous Rule – Battery Warning.

resumes its "normal" monitoring duty by selecting and applying preplanned rules.

Multiple parallel states and overlapping rules: While preplanned rules are applied according to the current system context, ubiquitous rules can apply any time that the respective event is triggered. This means that multiple rules may apply simultaneously. For example, a UAV might enter a no-fly zone whilst simultaneously experiencing a critical battery state. These two events trigger conflicting rules. The first rule increases the required period of GPS and altitude data, whilst the second decreases monitoring periods to preserve power. To address these conflicts, we provide a multi-layered resolution strategy. We explicitly specify rules for common combinations of states by specifying monitoring rules for those combinations. Either a detected trigger event or the transition out of an existing state causes the monitoring infrastructure to check rules that apply to all combinations of active states. We utilize a salience-based approach, as commonly adopted by other rules engines (e.g., Drools (Red Hat, 2022)); however, whereas they execute rules sequentially according to their salience, we only execute one selected rule to ensure that lower-priority rules do not override the monitoring behavior specified in selected higher-priority rules.

If an exact matching combination is found, then that rule is applied. Otherwise, rules are prioritized by scoring them according to the number of matches as follows:

$$score_r = 1 - \frac{|T_r \setminus A| * w_a + |A \setminus T_r| * w_b}{|T_r \cup A|} \quad (1)$$

where T_r denotes the set of triggers that are specified for the rule (i.e., the two trigger Events `battery_critica` and `prohibited_airspace`) and A denotes the set of ubiquitous states that are currently active. The score calculates the ratio of rule triggers versus active states. In addition, w_a and w_b define penalty weights when a state is active but not in the rule trigger, or vice versa, a rule has an additional trigger specified that is not currently active. The rule with the highest score is selected and ties are broken using a salience element indicating the priority of a rule.

4.2. Default values and assumptions

Our DSL supports the definition of two additional elements: Default Values and Assumptions.

Default Values: To make rules more precise and avoid the need for specifying long lists of properties in each rule, we support *Default* value definitions.

```
Default 'Drone.State' off;
Default 'Drone.StartupChecks' keep;
```

Listing 3: Default specification for Monitoring properties.

Default values apply globally across all rules for specific properties (cf. Listing 3) and can assume one of two values. If a value is `off`, then unless a rule explicitly specifies a monitor

for that event, the monitoring is automatically turned off. If a default value is `keep`, then the monitor for this event remains unchanged and retains the period and scope from the previously executed rule. Introducing the default value concept allows to greatly reduce the number of monitors that need to be specified as part of each rule.

Assumptions: Finally, *Assumptions* are used to specify properties, such as the assumed monitoring period, that should not be violated by the monitoring rules. For example, Listing 4 specifies that the minimum period with which we can expect updates is 500 ms for `Drone.State` messages and 1 s for `Drone.Startupchecks`.

```
1 Assumption 'Drone.State' MIN_PERIOD: 0.5 seconds;
2 Assumption 'Drone.StartupChecks' MIN_PERIOD: 1 seconds;
3 Assumption on Constraint 'separation_distance'  scope central
4         requires monitor 'Drone.Location';
```

Listing 4: Assumption specification for Monitoring properties.

Assumptions can be used to specify limitations on the properties' minimum periods and scopes. The minimum period specifies how often the monitored system guarantees updated values. If an assumption specifies a minimum period of x for a property and a monitoring rule requires the property to be monitored more often than that (with a period of $y < x$), the assumption is violated. With the assumptions from Listing 4, a subsequent rule specifying a period of, for example, 0.25 s for `Drone.StartupChecks` would be invalid, as it is not guaranteed that that data can be provided frequently enough by the system. We validate assumptions through a combination of static rules validation (see Section 5.3) and additional runtime checks.

5. Framework implementation

In this section, we present the framework implementation using Eclipse. Our approach is designed to support the end-user without requiring them to develop custom implementations for each monitoring rule. Therefore, our DSL is complemented by a code generation component that transforms monitoring rules into executable code that can be deployed to specific devices.

5.1. Generating monitoring components

User-defined rules, assumptions, and default values from the DSL are used in conjunction with the specified state transition diagram to generate executable code for the target system. This includes an API for the monitored system or device, and an adaptation component that handles transitions between preplanned states, rule execution, and the selection of ubiquitous rules. We aim to shift the burden of providing the right monitoring information from the system developer to the adaptive monitoring component.

To achieve this goal, the generated monitoring API provides methods for submitting monitoring data, regardless of the required period, while the internal adaptation mechanism executes actions, such as controlling the frequency, as specified in the rules. The mechanism leverages an internal state machine, generated from the state transition diagram and *monitoring executors* that are dynamically triggered. For example, as soon as the UAV is activated, it starts sending location data every 500 ms to the monitoring API. Based on the currently active state/context of the system, a monitoring executor is spun up that performs the actions specified in the active rule. Examples include forwarding the message every 0.5 s to the central server, in the `Preflight_Data` context, or discarding the message in the `Battery_Critical` context (as the monitor in this rule is set to the state *OFF*). We provide additional details and examples for the code generation as part of the evaluation in Section 7.1.

5.2. Prototype implementation

We use the Eclipse Xtext (Eclipse Foundation, 2021) and Xtend (Eclipse Foundation, 2022) frameworks, for the adaptive monitoring DSL and code generator which enables context assist and syntax highlighting. Static validation can also be directly performed inside the rule editor, and OCL constraints are automatically evaluated when a new rule is added or modified (cf. Section 5.3), and violations are directly reported within the Eclipse IDE. State transition diagrams are created using Papyrus, a UML editor extension for Eclipse (Lanusse et al., 2009). However, any tool that provides the diagram in a machine-readable format, e.g., XML or JSON, could be used. Xtend is then used to generate executable monitoring code for a specific target system. As part of the evaluation, we have created two code generators for a Java-based system, as well as a ROS-based Python system (cf. Section 7.1).

5.3. Static validation

Rule validation provides support for writing valid and consistent rules, so that users receive immediate feedback while using the DSL. While Xtext helps to ensure that the specified monitoring rules are syntactically valid, we created a dedicated Rule Validator component, in order to further improve the static validation. Our Rule Validator automatically checks a set of constraints and allows users to define declarative constraints in Java that are reflectively invoked at run time, and also allows them to describe a majority of our constraints in OCL (Cabot and Gogolla, 2012).

Scope Consistency: Different constraints can be specified using our DSL, such as indicating that a particular form of analysis requires state data to be analyzed centrally. The corresponding OCL constraint (`CentralScopeMismatch` in Listing 5) checks whether any constraint requires a particular property to be centrally monitored, whilst a conflicting rule sets the property's monitoring scope to local.

Assumption Consistency: The OCL constraint (`Assumption Frequency` in Listing 5) allows assumptions to specify the minimum period at which a property is reported. The constraint is violated if any rule sets the period to a lower value.

Coverage of Monitoring Properties: This constraint checks that all monitoring properties of the monitored system appear in at least one rule (i.e., important properties are not overlooked). The idea is that monitoring properties that are considered important should be used in at least one rule. The validator iterates over all specified monitoring properties and checks whether there exists at least one rule that states how this property (e.g., the UAV's state) is monitored in the adaptive monitoring system.

Default Values: An additional rule checks whether monitoring properties without a default value explicitly appear in all monitoring rules. In addition, we ensure that monitoring properties that are set as `keep` are switched on by at least one rule.

Static validation rules can be modified as OCL constraints or by adding additional Java rules in our implementation of Xtext Validator.

6. Evaluation objectives

When evaluating our adaptive monitoring framework, we focus on two main aspects. First, we evaluate the general feasibility of applying AMon in realistic contexts, by focusing on the DSL's expressiveness and ease of use. Second, we evaluate its efficiency and scalability to large-scale environments.

```
-- if there is a constraint with a central scope, the property must not
   be local in any rule
inv CentralScopeMismatch:
 constraints->forAll(c | c.scope.scope->includes('central')
 implies not (rules.monitors->exists(m | c.monitors->includes(m.name)
 and  m.changes->select(oclIsTypeOf(ChangeScope)).oclAsType(ChangeScope)
 ->exists(scope.scope->includes('local')))))

-- if a min frequency of x seconds is assumed, then there should
  not be any rule that sets the frequency < x
inv AssumptionFrequency:
 contracts->forAll(c  rules.monitors->forAll(m  m.name = c.name implies
 m.changes->select(oclIsTypeOf(ChangeFrequency)).oclAsType(ChangeFrequency)
 ->forAll(frequency.toReal()>=c.frequency.frequency.toReal())))
```

Listing 5: Excerpt of our OCL Constraints.

The first part of the evaluation assesses the general feasibility of our approach by creating adaptation rules, implementing code generators, and generating monitoring components for two different systems. We further perform realistic simulations of a UAV system, focusing on the approach's scalability and flexibility for use in a completely different system based on TurtleBot3 robots. Our research questions are as follows:

RQ1: Is AMon's DSL sufficiently expressive for specifying adaptive monitoring rules for real-world systems?

To answer this research question, we used AMon's DLS to specify rules for two different systems. First, Dronology (Cleland-Huang et al., 2018) a CPS for managing, controlling, and executing missions for UAVs, and second, TurtleBot3 Robots (Park and Son, 2021), using the ROS-based hardware and software robotics platform. For each system, we derived two scenarios. We then created all necessary artifacts, including state transition diagrams, monitoring adaptation rules in our DSL, and a code generator for a Java-based API for Dronology and a Python-based ROS connector for the TurtleBots (Section 7.1).

RQ2: To what extent can AMon support the efficient and fine-grained adaptation of monitoring behavior, in comparison to a non-adaptive monitoring approach?

To answer the second research question and further assess the feasibility of the framework, we used the previously created rules and performed a series of simulations with a high-fidelity simulator that was provided as part of the Dronology system. Additionally, we use the TurtleBots to demonstrate the feasibility on real hardware (Section 7.2). For both systems, we compared our adaptive monitoring solution with a non-adaptive monitor in which data was processed as it was received from the system. The goal was to demonstrate that AMon can reduce the monitoring data that is sent to the monitoring framework according to the specified rules.

RQ3: Does the performance of AMon scale when handling a realistic number of rules and large quantities of monitoring data? To demonstrate the scalability of our approach, that AMon can handle rules for various different states and rules, we conducted experiments, scaling up the number of messages and ubiquitous rule combinations, to ensure that selecting the correct rules at runtime was still possible within a reasonable amount of time (Section 7.3).

Besides evaluating the general feasibility and scalability of our framework, we further investigated the *understandability* as well as *ease of use* of our DSL. This specifically pertains to creating new monitoring rules using our DSL, and how well potential end-users are able to understand the various concepts (rules, assumptions, etc.,) of our DSL and monitoring requirements encoded in these rules. For this reason, we conducted a user study (Section 8) with six participants providing them with a scenario drawn from our two example systems and evaluating their ability to create new rules for a predefined set of monitoring requirements. In a second step, we also assessed understandability of created rules without deep domain-expert knowledge about the system. For this purpose, we defined the following two research questions:

RQ4: How easy is it for users to define new monitoring behavior for a given scenario with the domain-specific language? Based on the previously derived scenarios for the Dronology and TurtleBot system, we asked participants to create a set of preplanned and ubiquitous rules, as well as assumptions and default values using our DSL.

RQ5: How understandable are the rules and the adaptations encoded in these rules? We investigated how understandable our DSL is by assessing how our participants comprehend the purpose of adaptation rules written in our DSL.

7. Experimental evaluation (RQ1–RQ3)

To support our planned experiments, three of the authors created preplanned and ubiquitous rules for four use cases. These included two unique use cases, based on a real-world application scenario drawn from literature and application examples for each of our two targeted systems.

Our first targeted application was Dronology, representing a UAV management control system with a publicly available set of use cases (Cleland-Huang et al., 2020; sUAS Use Cases, 2022). Dronology is a Java-based system with available source code for coordinating, planning, and flying missions with either physical or simulated UAVs. For the purpose of the evaluation, we chose two diverse use cases, one in which multiple UAVs contribute to a River Search and Rescue mission (UC1-RESC) and one in which a UAV performs Item Delivery (UC2-DELI).

To demonstrate that AMon can be applied to different systems using diverse technologies, we used TurtleBot3 robots for our second application. TurtleBots are small robotic systems, frequently used for research (Li and Tu, 2021; Mainampati and Chandrasekaran, 2021). In contrast to the Dronology system, they are based on the Robot Operating System (ROS) and applications for controlling them are mainly implemented in Python. Our first use case involved maneuvering through a narrow tunnel and detecting an obstacle (UC3-OBST) (cf. Fig. 3), whilst the second involved picking up and transporting an item (UC4-TRAN).

7.1. RQ1 – Expressiveness

With this first research question, we evaluated the general feasibility of our approach and assessed the expressiveness of our DSL. As part of this RQ, we investigated how different monitoring needs can be expressed in our DSL and if monitors can be generated based on the different adaptation rules.

For this purpose, we used the four use cases described above, selected relevant system states, created state transition diagrams, derived monitoring adaptation rules, and generated monitoring code.

Use Case Vignette: Tunnel Object Reporting
Description:
A fleet of TurtleBots drives through a tunnel and reports on the number of objects placed inside the tunnel.
Primary Actor: TurtleBot Operator **Trigger:** The TurtleBot Operator activates the mission.

Main Success Scenario:
1. **The TurtleBots are activated and start up.** [rule PR-1: pre setup]
2. **The TurtleBots calibrate its sensors and lidar unit.** [rule PR-2: calibrating]
3. **The TurtleBots are calibrated and ready for a mission.** [rule PR-3: ready for mission]
4. **One TurtleBot is indicated as the leader of the mission.**[rule PR-10: init mission]
5. **The fleet starts moving. All TurtleBots that are not leaders ensure a correct distance to the previous bot.** [rule PR-11: collaboration start]
6. **When an object is, the TurtleBot switches to pushing mode.** [rule PR-12: pushing]
7. **The operator confirms the correct identification of the object and the TurtleBot continues with its mission.** [rule PR-13: object confirmed]
8. **When all TurtleBots have left the tunnel, they position themselves in the final position.** [rule PR-14: final position]

Fig. 3. Use Case 3 – TurtleBot Obstacle Tunnel Search and Reporting (All use cases with their respective rules can be found in our GitHub repository).

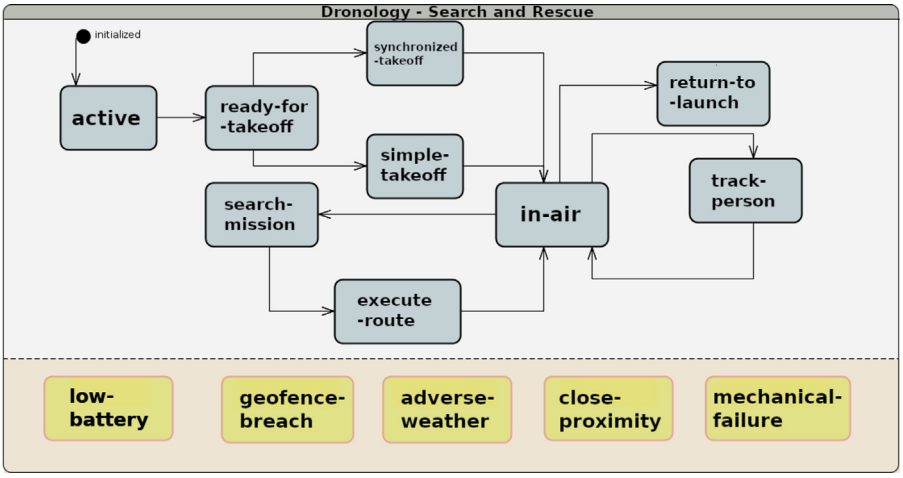

Fig. 4. State Transition Diagram for the Search and Rescue use case.

7.1.1. Specifying monitoring behavior

We created formal use case descriptions using a commonly adopted template (Cockburn, 2000) to specify the main success scenario describing the normal course of actions, as well as a set of exception states. These steps were based on the existing use cases for the Dronology system (Cleland-Huang et al., 2020) and application examples and tutorials provided for the TurtleBot3 robots (ROBOTIS, 2022).

Based on these use cases, we created a state transition diagram using Papyrus, representing both the normal and error states. In a second step, for each use case, two researchers created ubiquitous and preplanned monitoring rules and specified default values and assumptions. This resulted in a total of 21 states (17 normal course and 5 error states) and 15 monitoring adaptation rules specified with our DSL (10 preplanned and 5 ubiquitous rules) with some states and rules shared across both use cases (cf. Table 1). Fig. 4 provides an overview of the state transition diagram for RESC, the search and rescue use case. In RESC, multiple UAVs perform a synchronized, concurrent takeoff, meaning that frequent data regarding each UAV's location and altitude are needed, as well as additional information about the GPS accuracy, based on the number of satellites. Once all UAVs are successfully launched and commence flights to their designated search areas, computing power of the onboard computer and bandwidth can be freed up for other tasks, such as image recognition (cf. rule DR_PR-4 in Table 1).

For the TurtleBot use cases, we followed a similar approach and created two state transition diagrams containing 25 states (with 17 normal course and 8 error states) and then derived rules for each step of the use case, resulting in 14 preplanned and 4 ubiquitous rules. All rules are shown in Table 2.

For example, if the TurtleBot navigates to its target location (cf. TB_PR-7 in Table 2), information about its environment (e.g., odometry data), as well as information about the robot itself (e.g., velocity), are needed more frequently. In contrast, once the TurtleBot has reached the target location and is therefore no longer in motion, resources can be freed up for other tasks, such as determining whether the pickup has been completed.

7.1.2. Monitoring code generation

In order to generate actual monitoring code for the systems, we implemented two code generators. One for the Dronology system that generates executable Java Code and a second one for the TurtleBot robots that generates Python code and can then directly be integrated into the ROS-based robot application. For both code generators, we leveraged the Eclipse tool support, using Xtend in conjunction with the Xtext framework. It is important to note that the code generators are not system-specific, but rather technology-dependent, meaning that the Dronology code generator can be easily reused for generating monitoring code for other Java-based systems. The same applies to the TurtleBot code generator, requiring only minor adjustments to the system-specific parts (e.g., the concrete topics from which data is collected), which can be reused for other ROS-based applications.

Listing 6 provides a snippet of the resulting Python code showing how the property "BatteryStatus" is monitored. For each

Table 1
Preplanned and ubiquitous rules for the Dronology RESC and DELI use cases.

Rule		Use case	
		RESC	DELI
	Preplanned rules		
DR_PR-1	*Preflight Data*	✓	✓
DR_PR-2	*Awaiting Takeoff*	✓	✓
DR_PR-3	*Multi-UAV Synchronized Takeoff*	✓	
DR_PR-4	*Flying to Target Location*	✓	✓
DR_PR-5	*Searching for Victim*	✓	
DR_PR-6	*Active Tracking*	✓	
DR_PR-7	*Return to base*	✓	✓
DR_PR-8	*Single UAV Takeoff*		✓
DR_PR-9	*Drop-off location reached*		✓
DR_PR-10	*Drop-off in progress*		✓
	Ubiquitous rules		
DR_UB-1	*UAV Low Battery*: The UAV's battery is low; power needs to be preserved to safely return home	✓	✓
DR_UB-2	*UAV in no fly zone:* The UAV enters a prohibited area, additional monitoring data should be collected and logged centrally	✓	✓
DR_UB-3	*UAV Low Battery AND Close Proximity:* The UAVs battery is low, but at the same time a second UAV is in close proximity, i.e., only certain properties can be send at a reduced frequency	✓	✓
DR_UB-4	*Mechanical Failure:* A potential hardware failure might have occurred severely hampering the UAVs flying capability	✓	✓
DR_UB-5	*Adverse weather:* The UAV flies in rough weather conditions, more frequent updates on the UAVs state and location should be collected	✓	✓

Table 2
Preplanned and ubiquitous rules for the TurtleBot TRAN and OBST use cases.

Rule		Use case	
		TRAN	OBST
	Preplanned rules		
TB_PR-1	*Pre-Setup Data*	✓	✓
TB_PR-2	*Calibrating*	✓	✓
TB_PR-3	*Ready for Mission*	✓	✓
TB_PR-4	*In Home Zone*	✓	
TB_PR-5	*Travelling to Target Location*	✓	
TB_PR-5	*Entering Target Zone*	✓	
TB_PR-7	*Target Location Reached*	✓	
TB_PR-8	*Pick-Up Item*	✓	
TB_PR-9	*Return to Home Location*	✓	
TB_PR-10	*Initialize Mission*		✓
TB_PR-11	*Start Collaboration*		✓
TB_PR-12	*Object Pushing*		✓
TB_PR-13	*Object Confirmed*		✓
TB_PR-14	*Move to Final Position*		✓
	Ubiquitous rules		
TB_UB-1	*Turtebot Low Battery*: The TurtleBot's battery is low; power needs to be preserved to safely return to the home location	✓	✓
TB_UB-2	*Sensor Failure:* At least one of the sensors reports a system failure; more frequent data collection is advised to identify the cause of the failure	✓	✓
TB_UB-3	*Diagnostics Error:* The internal diagnosis detects an error in one of the hardware component requiring a more frequent data collection	✓	✓
TB_UB-4	*Pathplanning Error:* The specified navigation goal cannot be reached due to a pathplanning error	✓	✓

property that is monitored, a wrapper object and a set of methods are generated. When a state transition occurs, the respective configuration is activated and the monitoring thread is configured accordingly. The entire monitoring code can be automatically generated based on the state transition diagram and the monitoring configuration in our DSL. Similar code is generated for Dronology's Java implementation. In both cases, when a property is added, removed, or modified, no changes to the actual generators need to be made and the monitoring code can be regenerated automatically.

7.1.3. Analysis of results

By applying AMon to four use cases, we demonstrated that AMon can be used to specify adaptive monitoring rules for diverse states and scenarios. While creating state charts and specifying rules in the DSL requires some additional effort compared to using non-adaptive monitors, selecting and defining the rules in our DSL for the four use cases took three researchers approximately two person-hours per use case, where the majority of the time was spent on selecting appropriate states and deciding on the adaptive behavior. We found that the language was sufficiently expressive for the specification of rules with triggers and monitoring behavior. While some up-front investment is required to implement the code generator, this could be done in reasonable time, taking about 20 h to implement and test the generated monitoring code for each code generator. One observation we made when creating the respective rules was that, for a single system, with different use cases, a significant overlap

```python
def handle_BatteryStatus_data(): ### runnable.
  #...
   while not BatteryStatus_config_event.is_set():
      data = BatteryStatus_data['data']
      msg = ros_msg2json(data)
      # forward msg
      mqtt_forwarder_obj.publish('BatteryStatus', msg,
      BatteryStatus_config_obj['is_central_scope'])
      # wait according to config
   BatteryStatus_config_event.
   wait(BatteryStatus_config_obj['frequency'])
   BatteryStatus_config_obj['is_thread_running'] = False

def switch_to_state(state): ## state change
  match state:
      case 'active':
      # ... other states and properties
```

Listing 6: Example Python code generation for TurtleBot3 monitoring.

of rules, specifically exception cases (i.e. ubiquitous rules), as well as "common" system states. A more compositional approach (e.g., combining different rule fragments) could further reduce the effort and structure the rule space. Regarding RQ1, our adaptive monitoring DSL was able to capture the necessary rules for realistic use cases and we were able to generate monitoring code for the target system.

7.2. RQ2 – Efficient and fine-grained adaptation

The purpose of AMon is to facilitate the fine-grained definition of monitoring adaptation rules, so that relevant monitoring information can be collected, depending on the state or context of the system. To address RQ2 we evaluated whether context-specific monitoring can be used to reduce the number of messages and bandwidth when using AMon with customized adaptation rules. Based on the monitoring rules and code generator part of RQ1 we conducted a series of experiments. Concretely, we performed a series of simulations using the Dronology Software-in-the-loop (SITL) simulator with the *Item Delivery* use case (DELI), and conducted experiments with a TurtleBot robot applying the *Item Transportation* use case (TRAN).

7.2.1. Evaluation setup

Dronology — Item Delivery: We simulated multiple UAVs delivering items and compared the monitoring information received against a non-adaptive monitoring implementation. For this purpose, we used four Raspberry Pi 4 computers with 4 GB RAM and running Raspbian OS (32-bit, Debian 10). Three Raspberry Pi were set up with the Dronology Simulator and the latest version of ArduPilot SITL to simulate UAVs and their onboard computing capabilities, whilst the fourth Pi acted as the "central server" for receiving data. Each UAV was assigned a unique home location and tasked with five delivery runs to randomly selected locations within its vicinity. During each simulation, the UAV transitioned between the different states (flying, dropping item, etc.), executed the monitoring rules, and adapted the period and amount of data that was sent to the central server.

In the first run of the simulation, we assessed the preplanned rules and performed five "normal" delivery flights without any unanticipated errors. In the second run, we randomly injected an error during each delivery flight (e.g., a low battery warning, a proximity alert, or entering a no-fly zone) to trigger ubiquitous states, in order to validate the execution of ubiquitous rules. The third and fourth runs repeated the first two runs but without adapting the monitoring infrastructure regardless of its state and context. We executed each run three times and collected the *number of monitoring messages* that were sent for each property.

TurtleBot — Item Pickup: For the Item Transportation use case, we used a physical TurtleBot3 robot to compare our AMon adaptive monitoring implementation with non-adaptive monitoring. For navigation, we employed the navigation stack built into ROS, in conjunction with the pre-generated SLAM map. We implemented a small TurtleBot application using Python to start up, calibrate, and control the robot during the evaluation runs.

After starting the ROS nodes generated by AMon, we performed two evaluation runs. In each run, the TurtleBot was tasked with navigating to five target locations and returning to its home location after reaching a target goal (using SLAM). At a target location, the TurtleBot was loaded with a small item (a USB flash drive) to simulate the actual pick-up. During the run, AMon collected monitoring data with a local MQTT broker, that was set up on the edge device, and a central broker, running independently on a remote server. As with the Dronology case, we evaluated the pre-planned rules by performing five collection runs to different locations. We repeated the runs with adaptive monitoring disabled and collected data as originally published by the ROS components, regardless of their states and contexts.

7.2.2. Results

Dronology — Item Delivery: An overview of the evaluation results is presented in Table 3. For the preplanned scenario, the five simulated delivery flights performed by each UAV took approximately 32.5 min (which is within realistic flying capabilities of a physical UAV). In total, 6,521 messages were sent to the central monitoring server per iteration (average of three runs), and 136 transitions between different contexts were performed with monitoring rules being executed. Fig. 5 provides a partial overview of the changes in the period of the different monitors for one UAV.

Compared to the non-adaptive monitoring run where over 38,000 messages per run were collected, there was a significant reduction in messages that were sent to the central monitoring server. For the non-adaptive runs, no restrictions could be specified and the messages were sent "as is" when they were received from the UAV in the Dronology system. The most significant reduction was observed for the "FlightSchedule" messages (5750 messages vs. 171) that provided information about newly available routes. For the second run with the randomly injected errors, we observed that the UAVs switched from their "regular" state to the ubiquitous rules as soon as errors were injected. We again compared our adaptive approach with the non-adaptive approach and observed approximately 40,000 messages for the non-adaptive run vs. 7,200 messages for our adaptive approach, confirming that ubiquitous rules were triggered when an error was injected.

TurtleBot — Item Pickup: For the TurtleBot's item pickup use case, we executed three runs, with each run including five physical deliveries with and without our adaptive monitoring enabled. On average, adaptive runs took just under 16 min (15:50 min). They resulted in a total of 4,458 messages that were sent to the monitoring component via MQTT, and a total of 44 state transitions. This contrasted with the non-adaptive run where over 193,000 messages were reported. An overview of the results, with the different properties collected, are provided in Table 3 When using our adaptive monitoring approach, we were able to significantly reduce the total amount of messages sent, namely by more than 188,000. Most significant was the monitor for "MagneticField", where more than 99% (91,146 messages) of the initial messages were filtered. Furthermore, the non-adaptive monitors for "BatteryStatus", "JointState", "SensorState" and "Odometry" all sent messages with a frequency of about 23 Hz in the non-adaptive runs.

Table 3
Results of the DELI and TRAN use case simulation runs comparing our adaptive monitoring approach with a non-adaptive approach.

Scenario	Monitor	Non adapt. count	Adapt. mon. count
Dronology – Preplanned Rule Execution	Drone.Battery	3,829	1,258
	Drone.Startupchecks	5,747	99
	Drone.Status	5,747	1,349
	Drone.Location	5,714	2,459
	Drone.Monitoring	11,494	1,185
	Drone.FlightSchedule	5,750	171
Total		38,282	6,521
Dronology – Ubiquitous Rule Execution (Injected Errors)	Drone.Battery	5,780	1,235
	Drone.Startupchecks	5,780	100
	Drone.Status	5,780	1,052
	Drone.Location	5,780	2,793
	Drone.Monitoring	11,559	1,845
	Drone.FlightSchedule	5,784	172
Total		40,463	7,197
TurtleBot – Preplanned Rule Execution	Bot.BatteryStatus	21,453	337
	Bot.Velocity	9,419	694
	Bot.Diagnostics	944	504
	Bot.JointState	21,451	569
	Bot.MagneticField	91,404	258
	Bot.VersionInfo	934	179
	Bot.SensorState	21,455	492
	Bot.Odometry	21,284	780
	Bot.LaserScan	4,663	644
Total		193,007	4,458

7.2.3. Analysis of results

When performing simulation runs with the two systems using AMon, we observed a significant reduction in data that was sent to the monitoring framework. For all use cases, we found that specified preplanned and ubiquitous rules were executed correctly. Furthermore, whenever an error state for a UAV was introduced, the adaptive monitor for this UAV transitioned to its corresponding ubiquitous state and returned to the normal state once the error was removed. We conclude that AMon successfully adapted its monitoring behavior based only on the user-defined monitoring rules.

7.3. RQ3 – Scalability

With respect to scalability, we evaluated (i) that AMon can handle a large number of ubiquitous rule combinations where the most suitable rule needed to be selected and applied at runtime, and (ii) demonstrating the general suitability of our approach for a realistic robotic or CPS, handing a large number of messages. For the first part, we performed the rule selection evaluation on a Standard Intel Core i5 Laptop, with 16 GB of RAM, running Linux Mint (20.3), and for the second part, we executed additional simulation runs using the setup described in Section 7.2.1.

To ensure timely selection of rules and demonstrate the scalability of the rule selection and activation component, we created a new, larger rule set of 150 ubiquitous rules. To generate diverse triggers for each rule, we first created 20 ubiquitous states. Each rule's trigger was then defined as a random combination of either 1, 2, or 3 of these ubiquitous states, and a random salience level was added to each rule. Once the rule set was generated, we executed the rule selection component 1000 times, in each case randomly selecting one of the possible trigger combinations. Each execution performed was internally logged with AMon, and we measured the execution time and collected information about the selected rules and scores. After the executions were finished, we analyzed the results and selected rules.

The selection and rule execution time was 10 ms (median, 1st quartile: 8 ms, 3rd quartile: 11 ms). Out of the 1000 rule executions, we further spot-checked 50, to ensure that the correct rules were in fact executed, which was the case for all 50 rules checked. With regards to RQ3 we conclude that AMon is capable of handling both a large number of messages from different properties and different UAVs as well as many rules and ubiquitous states.

Additionally, to validate the scalability of message transmission, and ensure that our framework can in fact handle realistic amounts of monitoring data, we added 20 additional artificial monitoring properties to the Dronology system, integrated them into rules, and set the monitoring period for all properties to 1 s. We then re-executed our DELI Item Delivery simulation and collected the number of messages transmitted. For the approximately 32-minute run, we received 118,915 messages, i.e., more than 3,700 messages per minute, and observed a constant number of messages being received by the central server throughout the run without any observed increase in latency. In a second step, we scaled up the number of UAVs. We replicated the setup of the Raspberry Pi onboard computers in a Docker container and executed 20 parallel instances, i.e., 20 UAVs each sending 20 properties at a frequency of 1 s. We again executed the delivery simulation and collected over 793,000 messages and over 900 rule executions in the 32-minute run, again without any observable increase in latency.

7.3.1. Analysis of results

With regards to RQ3, for both dimensions we scaled up, message transmission and rule selection, we could observe that AMon was capable of handling large amounts of monitoring data and rules as input to the selection algorithm. Especially the latter is critical for adaptive monitoring, as ubiquitous rules are related to erroneous system behavior, and collecting required data can be crucial, e.g., for documenting errors, or detecting potentially dangerous situations before they occur. One aspect we observed is that depending on the different ubiquitous states might reflect different levels of severity, and it might be necessary to introduce more sophisticated mechanisms for specifying rule combinations and/or hierarchies (cf. Section 10).

8. User study (RQ4 and RQ5)

For our user study, we invited six participants to perform a series of tasks using our DSL. The participants were contacts in

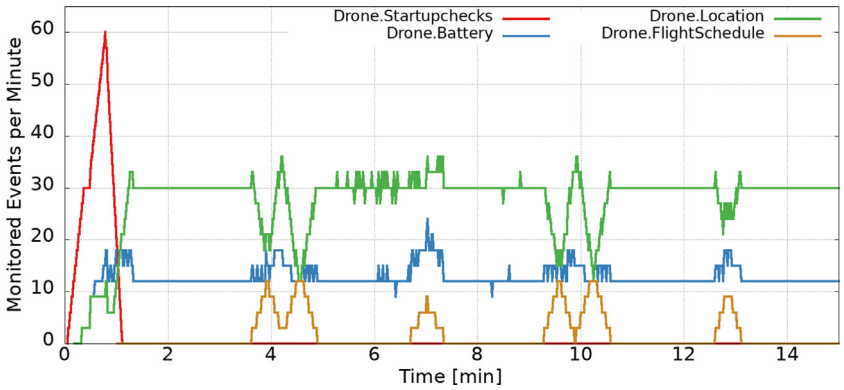

Fig. 5. Overview of monitored message per minute from one UAV in the Dronology system for different monitored properties.

our professional networks. The main selection criteria included for the participants to have basic knowledge about CPS and have actively worked with, or developed applications in the context of CPS. This was the case for all six selected participants, with experience either in the domain of industrial automation, UAVs, IoT, or smart devices. Furthermore, all participants had several years of general programming experience as well as CPS experience, and four participants have industry experience. None of the study participants was involved in the design or development of our framework and DSL.

8.1. Study setup

The study consisted of three parts: (1) A set of tasks where participants were asked to write monitor adaptation rules based on a description of the scenario; (2) a "Glitch Detector" task, where we presented participants with a number of monitor adaptation rules written in our DSL and asked them to find glitches/problems in the rules; and (3) a semi-structured interview in which we asked participants about their experiences when writing monitor adaptation rules, difficulties, and potential improvements.

To evaluate the ease of use, we asked the subjects to express monitoring adaptation rules in our DSL for either (i) the UAV delivery scenario, or (ii) the TurtleBot item transportation scenario. Each participant was presented with one of the two scenarios and was provided with a textual description of the rules that should be created for a specific use case. Each session took approximately 45–60 min (with one session taking slightly longer due to technical issues not related to the study) and started with a brief introduction of approximately 10 min. In the first part (Task 1), the participants were asked to write default values, assumptions, and two preplanned and two ubiquitous rules based on a description of the expected monitoring behavior.

To assess the DSL's understandability, we followed Hoffman et al.'s (Hoffman et al., 2019) recommendations for eliciting our subjects' mental models and analyzing their understanding. More specifically, we used a glitch detector task (Task 2), in which participants identify things that are wrong in a system/explanation. The glitch detector task allowed us to identify semantic issues in our DSL. For this task, we gave the participants 10 min to find examples of the following four errors in a set of rules:

1. *Assumption violation*: The frequency within a rule violates an assumption.
2. *Default value missing*: Given our assumption that if no default value is defined, the monitoring property needs to be defined in all rules.
3. *Entry vs. exit trigger*: A mismatch of whether a rule should be triggered when an entry or an exit event occurs.
4. *Local vs. central monitoring*: A mismatch of whether a property should be monitored locally or centrally.

All static validation rules were disabled in the IDE, as these would have directly generated error messages in the Eclipse Error View.

Finally, we conducted a post-study interview in which we asked participants several questions about their experience with writing and understanding rules, including what they liked and disliked about the approach, and whether they had any suggestions for improving the DSL. Furthermore, we collected demographic data about the participants and their background.

To validate the materials and tasks prior to running our study, we ran a pilot study with a Ph.D. student who was familiar with the domain. To improve clarity, we made minor adjustments to the tasks descriptions and study material. We used a think-aloud protocol to collect data from the different tasks, and participants were encouraged to express problems or issues they face whilst writing rules. One researcher collected important statements, which were later augmented by extracted statements from the transcribed recordings. For the semi-structured interview, we asked four Likert-scale questions and a series of open-ended questions, focusing on how well the participants understood the different concepts and if they had suggestions for improvements. After the interviews, two researchers consolidated the notes, extracted key statements, and documented the main findings of the results.

8.2. Study results

Table 4 provides an overview of all study participants and the results of the rule creation and glitch detector tasks. All six participants successfully completed the first tasks, which included creating default values and assumptions and writing preplanned and ubiquitous rules.

For the second task, the glitch detection, four out of six participants were able to detect all four errors, whereas the remaining two managed to correctly identify three errors, but missed one. In both cases, the error that was overlooked was related to an incorrect monitoring period, which was constrained by an assumption (i.e., the rule incorrectly specified a lower period of a property than the corresponding assumption).

When discussing this error, all participants stated that having static validation support available would significantly ease this task and reduce the effort of manually checking the rules every time changes were made. All participants found it generally easy to write rules based on the desired behavior, with one participant mentioning that they *"[...] like using the DSL and it is very easy to define rules in the editor"*, Both the preplanned and the ubiquitous rules were considered very easy to understand, with one participant stating that he *"[...] had no problems understanding what the*

Table 4
Overview of study participants, experience, and results for the two tasks. Task 1, the Creation of rules (T1), and Task 2, finding glitches in a set of predefined rules (T2).

Part.	UC	Exp [years]	CPS Experience	T1: Rule Cr.	T2: Glitch Det.
P1	TB	14/7	IoT, home automation, robotic systems	4/4	4/4
P2	TB	16/4	ind. automation, robotic systems	4/4	4/4
P3	TB	7.5/1.5	digital twins, production systems, ind. 4.0	4/4	3/4
P4	DR	12/6	IoT, security, microcontroller	4/4	4/4
P5	DR	8/4	drones, robots	4/4	3/4
P6	DR	14/1	industrial automation systems	4/4	4/4

rules should do", and the provided state transition diagram made it easy for them to relate specific monitoring behavior to certain states of the system.

One observation made during the study was that all participants extensively used copy&paste to first create a rule and then paste the finished rule multiple times and only adapt the values and trigger event. One participant mentioned that *"templates or a GUI where you can select a rule and then insert it"* would further ease the tasks of writing rules and reduce time to configure individual properties. Generally, all participants found the tool support useful, and the autocomplete feature for both the states and property names was positively mentioned by participants. We also received valuable comments regarding the DSL itself. Participants noticed minor inconsistencies in the language (e.g., the use of quotation marks, and upper and lowercase keywords). Most notably, participants mentioned that the difference between the two types of rules (Preplanned and Ubiquitous) was not immediately clear and required additional context and description to fully comprehend when each type of rule would need to be written. During the discussions with study participants, the terms "State-specific" (as these rules are specific for a certain state in the state chart), and "Global" emerged. We will use this feedback and comments as we evolve our AMon framework and DSL, with a specific focus on clarity, readability, and ease of use.

In summary, for RQ4, we can conclude that creating rules, assumptions, and default values was an easy and straightforward task and could be performed by participants with very limited training. With regards to RQ5, all six participants had no issues fully comprehending the provided rule file. Four participants identified all mismatches between the natural language description and the DSL rules. The error that was missed by two participants, was related to violations of specified assumptions. When discussing the error participants had no issue understanding the nature of the error and stated that it was simply overlooked due to the large number of rules. Our static validation component targets this issue and supports users in identifying potentially invalid configurations. As code generation is performed automatically, the most time-consuming task is the creation of the rules, as well as the selection of properties that need to be monitored.

9. Threats to validity

Internal validity. Our user study was designed to evaluate AMon's capabilities to specify monitoring rules. All participants had previous experience with CPS. Our participants might have been biased or impacted by the projects they were previously involved in. These factors might have confounded the effects we observed. Working with similar systems in the past can affect the ease of learning new approaches and expressing monitoring behavior in our DSL. We aimed to gain insights into how well the DSL and tool support can be used and if improvements to the DSL can be made. We decided to conduct a think-aloud study to broadly elicit participants' mental models and collected interview transcripts, participants' answers to glitch detector tasks, and the created rule files to triangulate data and improve internal validity.

External validity. External validity is concerned with the generalizability of our findings to other contexts. The results of our work might not generalize beyond the cases we considered. Our Dronology UAV management and control system informed the design of AMon and some of the reported challenges and resulting requirements for adaptive monitors are based on observations we made when designing Dronology. Additionally, we have applied AMon to TurtleBot robots from a different application domain, that uses different hardware, and a different technology stack. Based on our findings, we expect our approach to be applicable to other types of CPS with similar characteristics. In its current form, AMon is particularly suitable for systems consisting of multiple devices for which local and central monitoring behavior needs to be defined (as in a system of systems).

The initial implementation generated a monitoring component for the Java-based Dronology system, and we were able to extend it for a ROS-based system. Regarding data measurement, the results of our Dronology experiments are based on simulated UAVs. In terms of scalability, we considered a limited amount of rules and properties. However, our experiments with scaling the number of rules and events with generated data (RQ3) go far beyond what one would typically expect in practical scenarios, which demonstrates that our approach has the potential to scale to an adequately large number of rules and events. We have conducted several simulation runs as part of the experiment, three per scenario, and previous experiments with Dronology have demonstrated almost seamless transitions from the high-fidelity ArduPilot simulator (ArduPilot, 2022) to physical UAVs. In the future, we plan to perform additional simulations and field tests with physical UAVs.

Conclusion validity. Conclusion validity is concerned with relationships in the data that do not exist or missing relationships that should not have been reported. Our study does not aim at statistically significant conclusions and we aimed to elicit important factors broadly in our study. The reliability of our findings might have been compromised by the limited data collection in six 45-60 min sessions. Our selection criteria required all participants to have CPS experience and additionally, four out of six participants have additional industry experience. To strengthen conclusion validity, additional sessions can be performed with further participants. We described our study in detail and made the material available to facilitate replication of the study.

Construct validity. Construct validity is concerned with the degree to which the measures in our study correctly reflect real-world constructs. The participants might have behaved differently, given that they were observed during the study (evaluation apprehension). Moreover, we designed tasks to evaluate AMon under realistic conditions, but actual scenarios might vary in practice. To mitigate threats to construct validity, we aimed to be as transparent as possible about the concepts we were investigating, communicated the purpose of the study to our participants, carefully discussed the study setup and execution among multiple researchers, and conducted a pilot study.

10. Discussion

Our experiments applied AMon to two real-world CPS. We created adaptive monitoring rules and generated monitoring code for four use cases and two technologies (i.e., Java and Python using ROS). Our findings indicate that AMon can be used in heterogeneous environments. We could describe states relevant for adaptive monitoring in a state transition diagram and identify system events and data needed for checking runtime constraints. Our evaluation further demonstrates that the amount of data sent to the central monitoring server was significantly reduced compared to a non-adaptive monitoring environment, and that AMon scales to a large number of events and rules. The proposed run-time adaptation of monitoring rules facilitated by our DSL can support context-specific changes to the monitoring behavior. The effort and level of granularity, to a certain extent, depends on the level of detail a developer wants to specify properties at for each state. For example, if no specific period is defined then the default value or the one from the previous state is used. While adaptive monitoring work has focused, e.g., on adaptive sampling, our contributions focus on adjusting the monitoring behavior to the contexts of a system. Furthermore, the findings from our user study indicate that the DSL is easy to understand and can be used with little effort to clearly specify what and how parts of a system should be monitored. Based on these empirical findings, we summarize four lessons learned and avenues for further extensions.

• **Automation Support for Rule Creation and Maintenance:** Assumptions for constraints, which are currently created manually from the constraints in our prototype implementation, provide further automation potential. During the study, specifying default values and corresponding assumptions was regarded as somewhat tedious by participants. Additionally, we observed that all participants extensively used a copy&paste approach combined with the auto-completion features to write new rules and specify monitoring behavior. One participant commented that *"[...] introducing prefilled templates"* could significantly contribute to reducing the effort one has to put into writing rules for a system. One approach to tackle this issue could be to provide a combined graphical drag&drop editor, e.g., Google Blockly (Yamashita et al., 2017; Culic et al., 2015; Mao et al., 2019) that allows writing rules in our DSL and at the same time provides a graphical user interface for different users.

• **Additional Combination Features for Rules:** Our dual approach of combining rule triggers and selecting rules based on a prioritization algorithm, allowed us to define arbitrary combinations of ubiquitous events and to account for different situations. In our evaluation, we observed situations that would result in conflicting monitoring needs, e.g., when a low battery level raises the need to reduce monitoring whilst close proximity to other UAVs requires precise location data to be collected to avoid collisions. In these situations, and to reduce the burden on the user specifying monitoring periods manually for each state, an additional cost–benefit analysis could help to identify optimal monitoring rules or a combination of rules. Previous work has incorporated cost factors in cost-aware logging mechanisms (Ding et al., 2015) and leveraged cost–benefit analysis in the context of self-adaptive systems (Van Der Donckt et al., 2018).

• **Extended Operations for Describing Monitors and Assumptions:** Our DSL provides several options for describing monitors and assumptions. These include turning monitoring on and off, setting monitoring periods, and changing between local and central monitoring. However, in certain situations, the relatively simplistic rules might not be sufficient. For example, for a monitoring adaptation rule related to proximity to other UAVs, the monitoring period could be calculated based on the number of UAVs in the air and a scaled factor associated with the actual distances. This would not only allow adaptation of the period based on a state and a given value, but to continuously update that value based on the given formula. Therefore, defining additional aggregation functions, or temporal properties as part of the DSL adds additional flexibility for configuring monitors.

• **Runtime Reconfiguration of Monitors:** Depending on the nature of the system, one aspect that would further enhance adaptive monitoring capabilities is the "adaptation of the adaptation rules" at runtime. As pointed out by a study participant, *"[...] the operator of the drone system might also want to adapt monitoring behavior [during a mission]"*. With our approach, this could be easily supported, for example, in the context of a highly configurable ROS-based system. Not only would this allow adaptation of the monitoring behavior based on the predefined rules, but it would further enable a human to (temporarily) modify the adaptive monitoring behavior.

11. Related work

The need for adaptive monitoring relates to the importance of balancing the CPU resource consumption with the freshness of collected data, so that analyses could be effectively performed (Moui et al., 2012).

Existing monitored data has been used to reason about the system state in order to adapt the monitoring infrastructure and reduce the number of required monitors (Casanova et al., 2014). Brand and Giese proposed a generic adaptive monitoring approach based on analyzing queries on a runtime model and adjusting periodic or event-driven monitoring tasks (Brand and Giese, 2019); however, they require an existing architecture and implementation of a self-adaptive system on which to execute queries. In contrast, AMon is intended for use on any system regardless of its use of runtime models, and it generates code for diverse target systems. To support continuous adaptive monitoring, Brand and Giese also suggested requirements for a runtime model language (Brand and Giese, 2018, 2019). In their envisioned approach, runtime models include information about monitorable properties, as well as their results at run time, to allow the analysis of information needs and monitoring adaptation. Sakizloglou et al.'s approach (Sakizloglou et al., 2020) leverages runtime models to check for conditions of adaptation rules with temporal requirements. They aim to reduce the amount of monitoring information to a minimum by pruning the runtime model to only those parts needed for analysis. Their approach could be combined with AMon to also explicitly capture information about the contexts in which certain monitoring information is needed. Our work is more related to DYNAMICO (Tamura et al., 2013; Villegas Machado, 2013) which provides a separate monitoring infrastructure and supports users in defining monitoring properties and metrics. However, AMon goes beyond the scope of these adaptive monitoring approaches by integrating on-site vs. off-site monitoring. In the context of distributed CPS, we are not aware of other approaches that explicitly consider these aspects and enable users to specify adaptation rules in a lightweight fashion. The importance of monitoring self-adaptive applications within edge computing frameworks has been stressed in a state-of-the-art review (Taherizadeh et al., 2018), albeit with a strong focus on cloud applications. Given the increasing prevalence of computations performed on edge devices, we saw the need to address these issues in AMon.

Several monitoring approaches have been proposed for CPS and large-scale systems. For example, a task planning and execution monitoring framework uses temporal action logic to specify the behavior of the system (Doherty et al., 2009). For monitoring of UAVs, the ReMinDs framework has been developed (Vierhauser

et al., 2018). Other related approaches focus on synthesizing monitored autonomous systems (Machin et al., 2014). However, while some of these approaches facilitate system adaptation, none of them consider the effects of different states and conditions on the monitoring environment itself. Kieker (Ehlers and Hasselbring, 2011) provides capabilities for inserting probes to intercept the execution of system interface operations, application-level monitoring, system-level measurements, and CPU usage of an application. It also supports adaptation of monitoring rules for (de-)activating probes relevant to the current monitoring task. HiFi (Al-Shaer, 1999) uses programmable agents and filters to configure the infrastructure at run time and adapt agents.

To collect runtime data from a system, various adaptive sampling techniques (Bartocci et al., 2012; Ding et al., 2015) have been proposed. For example, an approach for weighted trace sampling relies on the clustering of execution graphs, with the goal to maximize the diversity of collected execution traces (Las-Casas et al., 2018). (Narayanappa et al., 2010) proposed "Property-Aware Program Sampling", a profiling technique that uses program slicing in the context of statistical sampling-based instrumentation. Similarly, GAMMA focuses on reducing monitoring and instrumentation overhead (Orso et al., 2002). One aspect of GAMME is the optimization of the placement of probes and splitting the monitoring tasks across different instances of the software. Similar to our approach, TigrisDSL (Mertz and Nunes, 2021) has been proposed, which is a domain-specific language for creating monitoring filters. While these approaches focus on data collection rather than on a fully-fledged monitoring solution, they could be a valuable starting point for automating the specification of monitoring frequencies, reducing the burden on the user to specify all properties manually in the DSL.

A model-based architecture for interactive run-time monitoring has been proposed (Hili et al., 2020) that leverages model-based techniques, model-to-model transformation, and automated code generation. Its focus lies on supporting runtime monitoring activities for real-time and embedded systems and it does not adapt monitors based on different states of the system. A related approach for tracking the behavior of self-adaptive systems (Reynolds et al., 2020) relies on provenance graphs and a runtime model to analyze and explain the runtime behavior of a system. In the domain of robot systems, a model-based framework (Corbato et al., 2020) adapts robot control architectures at runtime. It targets ROS-based systems and uses the MAPE-K loop to trigger reconfigurations of the system.

12. Conclusion

In this paper, we have presented a model-based approach for adaptive monitoring of distributed CPS deployed in increasingly heterogeneous environments. AMon consists of a DSL for specifying monitoring adaptation rules, mechanisms to statically validate those rules, and automated code generation for easy deployment and use of monitors. Our DSL supports the definition of monitoring behavior, taking both local and centralized monitoring into account, and helps users to specify consistent monitoring rules. Our evaluation using a UAV and a TurtleBot system indicates that the DSL is sufficiently expressive to describe realistic use cases with reasonable time and effort. Our simulations demonstrated that the monitoring framework can considerably reduce the number of events that are sent to the centralized monitoring system. As part of our future work, we will further improve the DSL and provide additional graphical user interface support for rule creation. Additionally, we plan to further extend AMon by adding additional functionality to the DSL for defining aggregation functions and temporal properties and facilitating the runtime adaptation of the adaptation rules.

CRediT authorship contribution statement

Michael Vierhauser: Conceptualization, Methodology, Software, Investigation, Resources, Writing – original draft, Project administration. **Rebekka Wohlrab:** Conceptualization, Investigation, Resources, Writing – original draft. **Marco Stadler:** Software, Verification, Investigation, Writing – review & editing. **Jane Cleland-Huang:** Verification, Writing – review & editing.

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

Supplementary material can be found in our repository: (Anon, 2022b) and Zenodo: http://dx.doi.org/10.5281/zenodo.6860667.

Acknowledgments

The work in this paper has been partially supported by the Linz Institute of Technology (LIT-2019-7-INC-316), the US National Science Foundation (NSF) under Grant #CNS-1931962, and by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

Al-Shaer, E.S., 1999. Programmable agents for active distributed monitoring. In: Proc. of the Int'L WS on Distributed Systems: Operations and Management. Springer, pp. 19–32.

AMon, 2022. AMon GitHub repository. https://github.com/LIT-Rumors/amon-public. (Accessed 01 July 2022).

ArduPilot, 2022. Ardupilot software in the loop simulation. https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html. (Accessed 01 July 2022).

Baresi, L., Pasquale, L., Spoletini, P., 2010. Fuzzy goals for requirements-driven adaptation. In: Proc. of the 18th IEEE Int'L Requirements Engineering Conf.. IEEE, pp. 125–134.

Bartocci, E., Grosu, R., Karmarkar, A., Smolka, S.A., Stoller, S.D., Zadok, E., Seyster, J., 2012. Adaptive runtime verification. In: Proc. of the Int'L Conf. on Runtime Verification. Springer, pp. 168–182.

Brand, T., Giese, H., 2018. Towards software architecture runtime models for continuous adaptive monitoring. In: Proc. of the 13th Int'l WS on Models@run.time. pp. 72–77.

Brand, T., Giese, H., 2019. Generic adaptive monitoring based on executed architecture runtime model queries and events. In: Proc. of the 13th Int'L Conf. on Self-Adaptive and Self-Organizing Systems. IEEE, pp. 17–22.

Cabot, J., Gogolla, M., 2012. Object constraint language (OCL): A definitive guide. In: Formal Methods for Model-Driven Engineering: 12th Int'L School on Formal Methods for the Design of Computer, Communication, and Software Systems. Springer, pp. 58–90.

Casanova, P., Garlan, D., Schmerl, B., Abreu, R., 2014. Diagnosing unobserved components in self-adaptive systems. In: Proc. of the 9th Int'L Symp. on Software Engineering for Adaptive and Self-Managing Systems.

Cleland-Huang, J., Agrawal, A., Islam, M.N.A., Tsai, E., Van Speybroeck, M., Vierhauser, M., 2020. Requirements-driven configuration of emergency response missions with small aerial vehicles. In: Proc. of the 24th ACM Systems and Software Product Line Conf.. pp. 1–12.

Cleland-Huang, J., Vierhauser, M., Bayley, S., 2018. Dronology: An incubator for cyber-physical systems research. In: Proc. of the 40th Int'L Conf. on Software Engineering: New Ideas and Emerging Results Track.

Cockburn, A., 2000. Writing Effective Use Cases, first ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Corbato, C.H., Bozhinoski, D., Oviedo, M.G., van der Hoorn, G., Garcia, N.H., Deshpande, H., Tjerngren, J., Wasowski, A., 2020. MROS: Runtime adaptation for robot control architectures. arXiv:2010.09145.

Culic, I., Radovici, A., Vasilescu, L.M., 2015. Auto-generating google blockly visual programming elements for peripheral hardware. In: Proc. of the 14th Int'L Conf. in Education and Research. IEEE, pp. 94–98.

Ding, R., Zhou, H., Lou, J.-G., Zhang, H., Lin, Q., Fu, Q., Zhang, D., Xie, T., 2015. Log2: A cost-aware logging mechanism for performance diagnosis. In: Proc. of the 2015 USENIX Technical Conference. pp. 139–150.

Doherty, P., Kvarnström, J., Heintz, F., 2009. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. Auton. Agents Multi-Agent Syst. 19 (3), 332–377.
Eclipse Foundation, 2021. Eclipse Xtext - language engineering framework. https://www.eclipse.org/Xtext. (Accessed 01 April 2021).
Eclipse Foundation, 2022. Eclipse XTend. https://www.eclipse.org/xtend. (Accessed 01 July 2022).
Ehlers, J., Hasselbring, W., 2011. A self-adaptive monitoring framework for component-based software systems. In: Proc. of the European Conf. on Software Architecture. Springer, pp. 278–286.
Erdelj, M., Natalizio, E., Chowdhury, K.R., Akyildiz, I.F., 2017. Help from the sky: Leveraging UAVs for disaster management. IEEE Pervasive Comput. 16 (1), 24–32.
Hili, N., Bagherzadeh, M., Jahed, K., Dingel, J., 2020. A model-based architecture for interactive run-time monitoring. Softw. Syst. Model. 1–23.
Hoffman, R.R., Mueller, S.T., Klein, G., Litman, J., 2019. Metrics for explainable AI: Challenges and prospects. arXiv:1812.04608.
Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Computer 36 (1), 41–50.
Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., Schnekenburger, R., Dubois, H., Terrier, F., 2009. Papyrus UML: An open source toolset for MDA. In: Proc. of the 5th Europ. Conf. on Model-Driven Architecture Foundations and Applications. Citeseer, pp. 1–4.
Las-Casas, P., Mace, J., Guedes, D., Fonseca, R., 2018. Weighted sampling of execution traces: Capturing more needles and less hay. In: Proc. of the ACM Symp. on Cloud Computing. pp. 326–332.
Li, K., Tu, H., 2021. Design and implementation of autonomous mobility algorithm for home service robot based on turtlebot. In: Proc. of the 5th Information Technology, Networking, Electronic and Automation Control Conf., Vol. 5. pp. 1095–1099.
Machin, M., Dufossé, F., Blanquart, J.-P., Guiochet, J., Powell, D., Waeselynck, H., 2014. Specifying safety monitors for autonomous systems using model-checking. In: Proc. of the Int'L Conf. on Computer Safety, Reliability, and Security. Springer, pp. 262–277.
Mainampati, M., Chandrasekaran, B., 2021. Implementation of human in the loop on the TurtleBot using reinforced learning methods and robot operating system (ROS). In: Proc. of the 12th Information Technology, Electronics and Mobile Communication Conf.. pp. 0448–0452.
Mallozzi, P., Nuzzo, P., Pelliccione, P., Schneider, G., 2020. CROME: Contract-based robotic mission specification. In: Proc. of the 18th Int'L Conf. on Formal Methods and Models for System Design. IEEE, pp. 1–11.
Mao, D., Wang, F., Wang, Y., Hao, Z., 2019. Visual and user-defined smart contract designing system based on automatic coding. IEEE Access 7, 73131–73143.
Mertz, J., Nunes, I., 2021. Tigris: A DSL and framework for monitoring software systems at runtime. J. Syst. Softw. 177, 110963.
Moui, A., Desprats, T., Lavinal, E., Sibilla, M., 2012. A CIM-based framework to manage monitoring adaptability. In: Proc. of the 8th Int'L Conf. on Network and Service Management. pp. 261–265.
Muccini, H., Sharaf, M., Weyns, D., 2016. Self-adaptation for cyber-physical systems: A systematic literature review. In: Proc. of the 11th Int'L Symp. on Software Engineering for Adaptive and Self-Managing Systems. pp. 75–81.
Narayanappa, H., Bansal, M.S., Rajan, H., 2010. Property-aware program sampling. In: Proc. of the 9th ACM SIGPLAN-SIGSOFT WS on Program Analysis for Software Tools and Engineering. pp. 45–52.
Orso, A., Liang, D., Harrold, M.J., Lipton, R., 2002. Gamma system: Continuous evolution of software after deployment. In: Proc. of the ACM Int'L Symp on Software Testing and Analysis. pp. 65–69.
Park, D., Son, W., 2021. ROBOTIS e-manual. ROBOTIS E-Manual, https://emanual.robotis.com/docs/en/platform. (Accessed 01 July 2022).
Pereira, E., Bencatel, R., Correia, J., Félix, L., Gonçalves, G., Morgado, J., Sousa, J., 2009. Unmanned air vehicles for coastal and environmental research. J. Coast. Res. 1557–1561.
Rabiser, R., Guinea, S., Vierhauser, M., Baresi, L., Grünbacher, P., 2017. A comparison framework for runtime monitoring approaches. J. Syst. Softw. 125, 309–321.
Rabiser, R., Schmid, K., Eichelberger, H., Vierhauser, M., Guinea, S., Grünbacher, P., 2019. A domain analysis of resource and requirements monitoring: Towards a comprehensive model of the software monitoring domain. Inf. Softw. Technol. 111, 86–109.
Red Hat, 2022. Drools. https://www.drools.org. (Accessed 01 July 2022).
Reynolds, O., García-Domínguez, A., Bencomo, N., 2020. Towards automated provenance collection for runtime models to record system history. In: Proc. of the 12th System Analysis and Modelling Conf.. pp. 12–21.
ROBOTIS, 2022. TurtleBot E-manual. https://emanual.robotis.com/docs/en/platform/turtlebot3/basic_examples. (Accessed 01 July 2022).
Sakizloglou, L., Ghahremani, S., Brand, T., Barkowsky, M., Giese, H., 2020. Towards highly scalable runtime models with history. In: Proc. of the IEEE/ACM 15th Int'L Symp. on Software Engineering for Adaptive and Self-Managing Systems. ACM, pp. 188–194.
Stocco, A., Weiss, M., Calzana, M., Tonella, P., 2020. Misbehaviour prediction for autonomous driving systems. In: Proc. of the ACM/IEEE 42nd Int'L Conf. on Software Engineering. pp. 359–371.
sUAS Use Cases, 2022. sUAS use case repository. https://github.com/SAREC-Lab/sUAS-UseCases. (Accessed 01 July 2022).
Taherizadeh, S., Jones, A.C., Taylor, I., Zhao, Z., Stankovski, V., 2018. Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. J. Syst. Softw. 136, 19–38.
Tamura, G., Villegas, N.M., Müller, H.A., Duchien, L., Seinturier, L., 2013. Improving context-awareness in self-adaptation using the DYNAMICO reference model. In: Proc. of the 8th Int'L Smyp. on Software Engineering for Adaptive and Self-Managing Systems. pp. 153–162.
Van Der Donckt, M.J., Weyns, D., Iftikhar, M.U., Singh, R.K., 2018. Cost-benefit analysis at runtime for self-adaptive systems applied to an Internet of Things application. In: ENASE. pp. 478–490.
Vierhauser, M., Cleland-Huang, J., Bayley, S., Krismayer, T., Rabiser, R., Grünbacher, P., 2018. Monitoring CPS at runtime-A case study in the UAV domain. In: Proc. of the 44th Euromicro Conf. on Software Engineering and Advanced Applications. IEEE, pp. 73–80.
Villegas Machado, N.M., 2013. Context Management and Self-Adaptivity for Situation-Aware Smart Software Systems (Ph.D. thesis). University of Victoria, p. 346.
Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M., 2013. On patterns for decentralized control in self-adaptive systems. Lecture Notes in Comput. Sci. 7475 LNCS, 76–107.
Yamashita, S., Tsunoda, M., Yokogawa, T., 2017. Visual programming language for model checkers based on google blockly. In: Proc. of the Int'L Conf. on Product-Focused Software Process Improvement. Springer, pp. 597–601.
Zavala, E., Franch, X., Marco, J., 2019. Adaptive monitoring: A systematic mapping. Inf. Softw. Technol. 105, 161–189.

Michael Vierhauser is a senior researcher at the LIT Secure and Correct Systems Lab at the Johannes Kepler University Linz, Austria. He holds a Master's degree in Software Engineering and Ph.D. in Computer Science from the Johannes Kepler University Linz. His current research interests include Cyber–Physical Systems, Safety Assurance, and Runtime Monitoring.

Rebekka Wohlrab is an assistant professor at Chalmers University of Technology in Sweden. She was previously with Carnegie Mellon University in the US and received her Ph.D. from Chalmers University of Technology. Her research interests include requirements engineering and software architecture, especially in connection with human-on-the-loop self-adaptive systems.

Marco Stadler is studying Business Informatics in the master's program at Johannes Kepler University Linz, with a focus on Networks & Security and Software Engineering. Prior to his time at JKU, he attended the bachelor's program in Business Informatics at the University of Regensburg, Germany. Besides his studies, Marco gained experience as a software developer in the IT finance industry. His research interests focus on runtime monitoring and Cyber–Physical Systems.

Jane Cleland-Huang is a Professor and Chair in the Department of Computer Science and Engineering at the University of Notre Dame. Her research interests focus upon Software and Systems Traceability for Safety–Critical Systems with a particular emphasis on the application of machine learning techniques to solve large-scale software and requirements engineering problems. She is the lead PI of the DroneResponse project, Chair of the IFIP 2.9 Working Group on Requirements Engineering, and Associate Editor of the Communications of the ACM. Recent organizational roles have included Program Chair of the 2020 International Conference on Software Engineering and General Chair of the 2021 International Requirements Engineering Conference.